CodeWarrior®

Targeting Mac OS

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 980829 ACD

Metrowerks CodeWarrior copyright ©1993–1997 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric Blvd., Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	Goto: `Metrowerks`

Table of Contents

1 Introduction **15**

Metrowerks Year 2000 Compliance 15
Read the Release Notes! 16
What's New in This Release 16
Multi-Monitor Debugging Support 16
Full Support for Mac OS Appearance Manager 16
Full Support for Mac OS Navigation Services 17
Online Help Support 17
CodeWarrior and Its Documentation 17
What's in This Manual 19
Where to Go from Here 21

2 Getting Started **23**

System Requirements 23
Installing CodeWarrior for Mac OS 24
CodeWarrior Compiler Architecture 24
Development Tools for Mac OS 25
CodeWarrior IDE 25
CodeWarrior Compiler for Mac OS 26
CodeWarrior Assembler for Mac OS 27
CodeWarrior Linker for Mac OS 27
CodeWarrior Debugger for Mac OS 27
Metrowerks Standard Libraries 28
The Development Process with CodeWarrior 28

3 Creating Application Projects for Mac OS **33**

Types of Applications 33
Creating a New Application Project 35
Creating Fat Applications 39
Creating Fat Applications with Multiple Targets 40
Creating Fat Applications with Subprojects 55
Choosing Libraries for Applications 62
A Note About SIOUX 63
68K C/C++ Libraries 63

Power Macintosh C/C++ Libraries 65
CFM68K C/C++ Libraries 66
68K Pascal Libraries 66
Power Macintosh Pascal Libraries 67
Setting Target Options 68
Setting the application's name 69
Setting the application's type and creator 69
Changing the memory requirements 70
Setting the stack size 71
Setting an application's 'SIZE' resource 71
Changing the startup code 73

4 Creating Static Library Projects for Mac OS 75
Types of Libraries 76
Creating a Library Project 77
Creating a 68K Mac OS Library 77
Creating a PowerPC Mac OS Library 80
Changing Library Project Options 83

5 Creating Shared Library Projects for Mac OS 87
Important Information on Shared Libraries 87
Types of Shared Libraries 88
Creating a Shared Library 89
Creating a CFM68K Shared Library 89
Creating a PowerPC Shared Library 90
Creating a Fat Shared Library 91
Exporting Symbols 92
Setting Shared Library Project Options 95
Choosing Libraries for a Shared Library Project 96
Difference Between Shared Libraries and Drop-Ins 98
Solving Common Problems 99

6 Creating Code Resource Projects for Mac OS 101
Types of Code Resources 101
Creating a Code Resource 103
Creating a 68K Code Resource Project 104

Creating a PowerPC Code Resource Project 107
Creating 68K Single or Multi Segment Resources 110
Creating a Single-Segment Resource Manually 111
Creating a Single-Segment Resource Automatically 111
Creating a Multiple Segment Resource 113
Setting Code Resource Options 114
Setting the Resource Names 115
Setting the Resource File Type and Creator. 116
Setting the Resource Type and ID. 116
Setting the Resource Attributes. 117
Choosing Where to Save the Resource File 119
Choosing Libraries for Code Resource Projects 121
Libraries For 68K Code Resource Projects 121
Libraries For PowerPC Code Resource Projects. 122
Writing Code Resources 123
Setting Up Globals in 68K Code Resources. 123
Creating Custom Headers for 68K Code Resources 124
Calling PowerPC Code Resources 128

7 Creating MPW Tool Projects for Mac OS 131
Creating an MPW Tool Project 131
Creating a 68K MPW Tool Project. 131
Creating a PowerPC MPW Tool Project 136
Creating a Fat MPW tool 140
Choosing Libraries for an MPW Tool Project 147
Writing an MPW Tool 150
Running the MPW Tool 152

8 Adding Project Resources 155
Adding Resource Files to a Project 155
Adding Rez files to a Project 156
Placing a Resource File in the Project Folder 158
Creating a Rez file from a Resource File 159

9 Mac OS Merge Settings 163
How to Get to Mac OS Merge 163

Mac OS Merge Settings Panels 165
Mac OS Merge Target Settings Panel 166

10 Target Settings for PowerPC 169
Target Settings Overview 169
Settings Panels for PowerPC 171
Target Settings 172
PPC Target 175
PPC Asm 183
Rez 185
PPC Processor 187
PPC Disassembler 192
PPC Linker 195
PPC PEF 199
Debugger Settings 204

11 C and C++ for PowerPC 207
Number Formats for PowerPC 208
PowerPC Integer Formats 209
PowerPC Floating-Point Formats 210
Calling Conventions for PowerPC 210
Register Variables for PowerPC 212
Optimizing Code for PowerPC 213
Register Coloring 213
Pragmas for PowerPC 214
Linker Issues for PowerPC 215
Deadstripping Unused Code and Data 216
Link Order 216

12 Pascal for PowerPC 217
PowerPC Notes Overview 217
Data Type Formats for PowerPC 217
Integer Types for PowerPC 218
Floating Point Types for PowerPC 220
Array Types and PACKED Arrays for PowerPC 222
Character String Types for PowerPC 222

Sets for PowerPC 223
Enumerations for PowerPC 224
Records for PowerPC 224
Calling Conventions for PowerPC 225
Overview of Calling Conventions for PowerPC 225
Parameters and PowerPC Calling Conventions 225
Registers and PowerPC Calling Conventions 226
Floating-point Values and PowerPC Calling Conventions . . 226
Functions and PowerPC Calling Conventions 227

13 Inline PowerPC Assembler 229

Working With Inline Assembly for PowerPC 229
Inline Assembler Syntax for PowerPC 230
Special PowerPC Instructions 231
Defining an Assembly Function for PowerPC 232
Creating Labels for PowerPC Assembly 234
Using Comments in PowerPC Assembly 234
Using the Preprocessor in PowerPC Assembly 235
Using Local Variables and Arguments in PowerPC Assembly 235
Creating a Stack Frame in PowerPC Assembly 236
Specifying Operands in PowerPC Assembly 237
Assembler Directives for PowerPC 240
entry 240
fralloc 241
frfree 241
machine 242
smclass 243
Intrinsic Functions for PowerPC 243
Low-Level Processor Synchronization 244
Floating-Point Functions 245
Byte-Reversing Functions 245
Setting the Floating-Point Environment 246
Floating-Point Instructions for the 603 and 604 246
Rotating the Contents of a Variable 246

14 Target Settings for 68K **249**
Target Settings Overview 249
Settings Panels for 68K 251
Target Settings 252
68K Target 255
68K Target, Application 256
68K Target, CFM68K Application 261
68K Target, Code Resource 263
68K Target, Library 269
68K Target, CFM68K Shared Library 270
68K Target, MPW Tool 272
Rez 275
68K Processor 278
68K Disassembler 288
68K Linker 291
CFM68K 294
Debugger Settings 298

15 C and C++ for 68K **301**
Language Extensions for 68K 302
Inline Data 303
Specifying the Registers for Arguments 304
PC-Relative Strings 305
Number Formats for 68K 306
68K Integer Formats 306
68K Floating Point Formats 307
Calling Conventions for 68K 309
Variable Allocation for 68K 310
Register Variables for 68K 311
Optimizing Code for 68K 312
Register Coloring 312
Generating Code for Specific 68K Processors 313
Generating Code for the MC68000 314
Generating Code for the MC68020 314
Generating Code for the MC68881 315

Hazards of 68020 and 68881 Codegen 317
Pragmas for 68K 319
Linker Issues for 68K 320
Deadstripping Unused Code and Data 320
Link Order 320

16 Pascal for 68K 323
Language Extensions for 68K. 324
The INLINE Directive. 324
Data Type Formats for 68K. 325
Integer Types for 68K 326
Floating Point Types for 68K 328
Array Types and PACKED Arrays for 68K 330
Character String Types for 68K 330
Sets for 68K 332
Enumerations for 68K. 332
Records for 68K 332
Calling Conventions for 68K 333
Simple Type Parameters for 68K 333
Structured Type Parameters for 68K 334
Function Results for 68K 335
A Note About 68K Registers and Mac OS 338

17 Inline 68K Assembler 339
Working With Inline Assembly for 68K 339
Inline Assembler Syntax for 68K 340
Defining an Assembly Function for 68K 341
Creating Labels for 68K Assembly 342
Using Comments in 68K Assembly 342
Using the Preprocessor in 68K Assembly 343
Using Structures in 68K Assembly 343
Using Global Variables in 68K Assembly 343
Using Local Variables and Arguments in 68K Assembly . . . 344
Returning From a Routine in 68K Assembly 345
Assembler Directives for 68K. 346
dc 346

ds 346
entry 347
fralloc. 347
frfree 348
machine. 348
opword 349
Assembler Instructions for 68K 349

18 Debugger Setup 351
Debugger Components 351
Configuring the Cross-Debugger 353
Installing Mac OS Debugger Software. 353
Configuring Windows/Mac Debug Connectivity 355
Notes on Using the Cross Debugger 356
Debugger Preferences. 356
MetroNub Settings 357

19 Debugging for Mac OS 359
Debugging Concepts 359
Stopping Under Program Control 359
Debugging Shared Libraries (PowerPC) 361
Shared-Library Limitations 361
Debugging a Shared Library in Four Steps. 362
Debugging Code Resources 363
Debugging Requirements 363
Runtime Debugging Requirements 363
Debugging a Code Resource in Four Steps. 364

20 Advanced Debugging 367
Debugging C++ and PowerPlant 367
Object-Oriented Data Types and References 368
Debugging Objects Created With `new` 369
Debugging Static Constructors and Destructors 371
Viewing PowerPlant Source Code 372
Debugging Inline Functions 372

Debugging Template Classes . . . 373
Debugging C++ Exceptions . . . 374
Debugging SOM . . . 375
Viewing SOM Objects . . . 375
Debugging Threads . . . 378
Debugging Profiled Code . . . 379
Stepping into a profiled routine . . . 379
Stepping out of a profiled routine . . . 380
Effect of stopping on the profile results . . . 380

21 DebugNew 381

Setting Up DebugNew . . . 382
Using DebugNew . . . 383
Limitations . . . 383
Error Reporting . . . 384
Leak Detection . . . 384
Leaks.log . . . 386
Advanced Techniques . . . 387

22 Mac OS Support Files 391

Using Libraries . . . 391
Using Shared Libraries . . . 392
Shipping Binaries Built With Shared Libraries . . . 392
Import “weak” . . . 393
Initialize Before . . . 394
Applications As Shared Libraries . . . 394
Using Apple’s Universal Interfaces . . . 395
C/C++ Headers . . . 396
Pascal Interfaces . . . 396
Prefix File . . . 397
Runtime Libraries . . . 397
About Runtime Libraries . . . 398
Exception Handling . . . 398
Choosing a Runtime Library . . . 398
Specifying Entrypoints . . . 399
Building a Monolithic Application . . . 399

Building a Monolithic Shared Library or Drop-in 400
Building a Multi-Fragment Application 401
Building Multi-fragment Shared Library. 402
Building a Multi-Fragment Drop-in 403
Installing Initialization and Termination Entrypoints 404
Using ANSI C/C++ Libraries 404
Multiprocessor Development. 406

23 Troubleshooting 407

Programming Specific 407
BowelsOfTheMemoryMgr 407
IDE Specific 408
Debugger Specific 409
General Problems 409
The debugger won't launch 409
Debug does nothing 410
Solutions 410
DebuggerdoesntRegainControlaftermyExceptionHandlerisCalled 411
Falling into MacsBug 412
Crashing in the Drag Manager 412
The debugger crashes 413
`Debugger()` and `DebugStr()` 414
`SysBreak()` and `SysBreakStr()` 415
Pointer variables have a "Bus Error" value. 415
Debugger Error Messages 416
Working With Other Debuggers 418
Jasik's The Debugger 418
QC Deactivates 419
QC crashes when I try to use it 419
MacsBug stack crawl 420

24 Library Naming Conventions 421

Library Naming Conventions 421

Index **423**

1

Introduction

This manual describes how to use CodeWarrior to develop code for Mac OS.

The introduction includes the following sections:

- Metrowerks Year 2000 Compliance—summary of Metrowerks Year 2000 Compliance posted on web site
- Read the Release Notes!—where to go for critical, last-second details
- What's New in This Release—new features since the previous release
- CodeWarrior and Its Documentation—a general description of the CodeWarrior architecture and documentation
- What's in This Manual—a description of the contents of this manual
- Where to Go from Here—recommendations for further reading

Metrowerks Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating systems. Except as expressly described above, the Products, in them-

selves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit: `http://www.metrowerks.com/about/y2k.html`.

Read the Release Notes!

Before you use the CodeWarrior IDE or a particular tool, you should read the release notes. They contain important last-minute information about new features, bug fixes, and incompatibilities that *may not be included in the documentation*.

The release notes directory is always included as part of a standard CodeWarrior installation. The release notes directory is also located at the top level of the CodeWarrior CD.

What's New in This Release

The CodeWarrior IDE has many new features. The following are most notable:

Multi-Monitor Debugging Support

On Mac OS systems with more than one monitor, additional features are available to utilize multiple monitors during debugging. For more information on this new feature, refer to the "Configuring IDE Options" chapter of the *IDE User Guide.*

Full Support for Mac OS Appearance Manager

The IDE is now fully Theme-savvy on Mac OS systems with Appearance Manager 1.1 or later installed. Appearance Manager 1.1 is only available as part of Mac OS 8.5 (Allegro). If you work at Apple or you are an Apple seeding site, please test the IDE running under Allegro.

Full Support for Mac OS Navigation Services

The IDE now uses Navigation Services instead of StandardFile on Mac OS systems with Navigation Services 1.1 or later installed. Navigation Services 1.1 is only available as part of Mac OS 8.5 (Allegro). If you work at Apple or are an Apple seeding site, please test the IDE running under Allegro.

Online Help Support

The new online documentation has "How do I..." task-oriented support topics and easier navigation with improved viewers. To learn more about this new feature, refer to the *IDE User Guide.*

CodeWarrior and Its Documentation

CodeWarrior is a multi-host, multi-language, multi-target development environment. What does that mean?

Multiple hosts CodeWarrior runs on several different operating systems, including Windows and Mac OS. The features, human interface, and operation of CodeWarrior is very similar on all hosts.

Multiple languages You can use CodeWarrior to program in several languages, including C/C++, Pascal, and Java. Third-party compilers provide support for other languages, such as Fortran. The languages available to you depend upon the target for which you are developing software.

Multiple targets You can use CodeWarrior to write software for several different chips or operating systems. CodeWarrior products support programming for a variety of embedded processors and real-time operating systems, the Java Virtual Machine, and desktop operating systems such as Windows, Mac OS, and BeOS.

Most features of CodeWarrior apply regardless of your preferred host, language, or target. General features of CodeWarrior are described in other manuals, such as the *IDE User Guide* and *Debugger User Guide*.

However, each target has its own unique features. This manual describes those unique features.

For a complete understanding of CodeWarrior, you must refer to both the general documentation and the documentation that is specific to your particular target, such as this manual.

The documentation is organized so that various chapters in this manual are extensions of particular generic manuals, as shown in Table 1.1. For a complete discussion of a particular subject, you may need to look in both the generic manual and the corresponding chapter in this Targeting manual.

Table 1.1 CodeWarrior documentation organization

This chapter...	Extends...
Creating Application Projects for Mac OS Creating Static Library Projects for Mac OS Creating Shared Library Projects for Mac OS Creating Code Resource Projects for Mac OS Creating MPW Tool Projects for Mac OS Target Settings for PowerPC Target Settings for 68K	*IDE User Guide*
Debugging for Mac OS	*Debugger User Guide*
C and C++ for PowerPC Inline PowerPC Assembler C and C++ for 68K Inline 68K Assembler	*C Compilers Reference*
Pascal for PowerPC Pascal for 68K	*Pascal Compilers Reference*
Mac OS Support Files	*MSL C Reference* *MSL C++ Reference*

For example, to completely understand the C/C++ compiler, you need to know the information in the *C Compilers Reference* (which covers the C/C++ front-end compiler) and the information in the C

and C++ for 68K chapter in this manual, which covers the back-end compiler that generates target-specific code.

What's in This Manual

Table 1.2 lists every chapter in this manual, and describes the information contained in each. However, this manual only contains information specific to Mac OS software development. See "CodeWarrior and Its Documentation" on page 17 for a discussion of how these chapters relate to other CodeWarrior documentation.

Table 1.2 Contents of chapters

Chapter	Description
Introduction	this chapter
Getting Started	installation and setup instructions, and an overview of CodeWarrior
Creating Application Projects for Mac OS	how to create Mac OS applications
Creating Static Library Projects for Mac OS	how to create Mac OS static libraries
Creating Shared Library Projects for Mac OS	how to create Mac OS shared libraries
Creating Code Resource Projects for Mac OS	how to create Mac OS code resources
Creating MPW Tool Projects for Mac OS	how to create MPW Tools that can be run under MPW or ToolServer
Adding Project Resources	how to add resources to a Mac OS project and how to use Rez files in your project
Mac OS Merge Settings	discusses target settings and function of Mac OS Merge

Chapter	Description
Target Settings for PowerPC	how to control the compiler and linker
C and C++ for PowerPC	details of the back-end compiler for PPC Mac OS development
Pascal for PowerPC	details of the back-end compiler for PPC Mac OS development
Inline PowerPC Assembler	details of support for inline assembly in the PPC compilers
Target Settings for 68K	how to control the compiler and linker
C and C++ for 68K	details of the back-end compiler for 68K Mac OS development
Pascal for 68K	details of the back-end compiler for 68K Mac OS development
Inline 68K Assembler	details of support for inline assembly in the 68K compilers
Debugger Setup	includes information on how to setup the cross debugger
Debugging for Mac OS	includes details on debugger settings panels for Mac OS
Advanced Debugging	advanced Mac OS-specific debugging issues including Debugging SOM, Threads, and PowerPlant
DebugNew	how to use the DebugNew library to debug memory handling problems in your C++ and PowerPlant code
Mac OS Support Files	libraries provided with CodeWarrior for the Mac OS

Chapter	Description
Troubleshooting	troubleshooting information specific to the Mac OS
Library Naming Conventions	appendix discussing how runtime and MSL libraries are named for Mac OS

Where to Go from Here

All the manuals mentioned here are available as part of the CodeWarrior documentation included with your product.

If you're new to CodeWarrior:

- Look for the CodeWarrior core tutorials in the `CodeWarrior Documentation` directory.
- Read "Development Tools for Mac OS" on page 25.

For everyone:

- For general information about the CodeWarrior IDE and Debugger, see the *IDE User Guide* and the *Debugger User Guide*.
- For information specific to the C/C++ front-end compiler, see the *C Compilers Reference*.
- For information on Metrowerks' standard C/C++ libraries, see the *MSL C Reference* and the *MSL C++ Reference*.
- For information specific to the Pascal front-end compiler, see the *Pascal Compilers Reference*.
- For information on CodeWarrior's standard Pascal library, see the *Pascal Library Reference*.

For general information on Mac OS programming:

- *Discover Programming* can teach you how to program in C/C++ or Pascal. You can learn more about Discover Programming at:

  `http://www.metrowerks.com/discover/`

- ZoneRanger is a Mac OS utility that enables you to see how your code is managing memory. If there are problems, you can fix them before releasing a final product. See the *ZoneRanger* manual for further information.
- MW Profiler is a Mac OS tool that allows you to see where optimizations are required in your code to make it more efficient. See the *MW Profiler* manual for more information.
- If you are using PowerPlant, you will be interested in reading the *Inside PowerPlant* documentation.
- *Inside Macintosh* is a series of books published by Addison-Wesley for Apple. This series is the definitive reference for programming the Mac OS Toolbox. These documents are available in Acrobat (PDF) format from many of Apple's FTP sites. *Inside Macintosh* is also available in HTML at Apple's Developer Web Site:

```
http://devworld.apple.com/techinfo/techdocs/mac/
mac.html
```

- If you are coming from a Windows or other platform background, and are not sure how to get started with Mac OS programming, you should visit this web site:

```
http://developer.apple.com/macos/intro.html
```

2

Getting Started

This chapter gives you the information you need to get CodeWarrior installed and running. For new CodeWarrior users, this chapter provides a brief overview of the CodeWarrior development environment, as well as a description of the development process in CodeWarrior as compared to a command-line environment.

This chapter includes the following topics:

- System Requirements — hardware and software requirements
- Installing CodeWarrior for Mac OS — how to install the various tools
- CodeWarrior Compiler Architecture — an overview of how front-end and back-end compilers work together
- Development Tools for Mac OS — an overview of the CodeWarrior IDE and its components
- The Development Process with CodeWarrior — how programming in CodeWarrior compares to command-line tools

System Requirements

Macintosh Requirements

- A Macintosh-compatible computer with either a 68K or a PowerPC processor. A PowerPC processor is recommended.
- Mac OS System 7.1.2 or later.
- A minimum of 16 MB RAM. 24MB or 32MB RAM is preferred.
- A CD-ROM drive to install CodeWarrior software, documentation, and examples.

Installing CodeWarrior for Mac OS

Your first step toward developing software for your target is to install the CodeWarrior tools. The tools include a variety of components such as the IDE, debugger, plug-in compilers and linkers, standard libraries, runtime libraries and headers, and all necessary documentation.

The CodeWarrior Installer automatically installs all necessary components. It is *strongly* recommended that you use the CodeWarrior Installer to ensure that you have all the required files. If you have any questions regarding the installer, read the instructions built into the CodeWarrior Installer itself.

To start the installation process, double-click the icon for the CodeWarrior CD on the Macintosh desktop. Then double-click the icon for the CodeWarrior Installer, located at the root level of the CD.

CodeWarrior Compiler Architecture

A proprietary multi-language, multi-target compiler architecture is at the heart of CodeWarrior. Front-end language compilers generate a memory-resident, unambiguous, language-independent intermediate representation (IR) of syntactically correct source code. Back-end compilers generate code from the IR for specific targets. The CodeWarrior IDE manages the whole process.

As a result of this architecture, the same front-end compiler is used in support of multiple back-end targets. In some cases, the same back-end compiler can generate code from a variety of languages.

All compilers are built as plug-in modules. The interface between the IDE and compilers and linkers is public, so third parties can create compilers that work with CodeWarrior.

Once the compiler generates object code, the plug-in linker generates the final executable. Some targets have multiple linkers available to support different object code formats.

Development Tools for Mac OS

Programming for Mac OS is much like programming for any other target in CodeWarrior. If you have never used CodeWarrior before, the tools you will need to become familiar with are:

- CodeWarrior IDE
- CodeWarrior Compiler for Mac OS
- CodeWarrior Assembler for Mac OS
- CodeWarrior Linker for Mac OS
- CodeWarrior Debugger for Mac OS
- Metrowerks Standard Libraries

If you are an experienced CodeWarrior user, this is the same IDE and debugger that you've been using all along. You will, however, need to become familiar with the Mac OS runtime software environment.

CodeWarrior IDE

The CodeWarrior IDE is the application that allows you to write your software. It controls the project manager, the source code editor, the class browser, the compilers and linkers, and the debugger.

The project manager may be new to those more familiar with command-line development tools. All files and settings related to your project are organized in the project manager. This allows you to see your project at a glance, and eases the organization of and navigation among your source code files. The IDE manages all build dependencies.

A project may also have multiple "targets." In this context, a target is a separate build (with its own settings) that uses some or all of the files in the project. For example, you can have a debug version and a

release version of your software as separate targets in the same project.

For more information about how the CodeWarrior IDE compares to a command-line environment, see "The Development Process with CodeWarrior" on page 28. That short section discusses how various parts of the IDE implement the classic features of a command-line development system based on makefiles.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The CodeWarrior CD includes a C/C++ compiler for the Mac OS family of processors. Other CodeWarrior packages include C, C++, Pascal, and Java compilers for Win32, Mac OS, and other platforms.

For more information about the CodeWarrior IDE, you should read the *IDE User Guide*.

CodeWarrior Compiler for Mac OS

The CodeWarrior compiler for Mac OS is an ANSI compliant C/C++ compiler. This compiler is based on the same compiler architecture that is used in all of the CodeWarrior C/C++ compilers. When used with the CodeWarrior linker for Mac OS, you can generate Mac OS applications and libraries that conform to Apple's ABI guidelines.

For more information about the CodeWarrior C/C++ language implementation, you should read the *C Compilers Reference*. For information on the Mac OS back-end compiler, see "Target Settings for PowerPC" on page 169, "Target Settings for 68K" on page 249, "C and C++ for PowerPC" on page 207, and "C and C++ for 68K" on page 301.

CodeWarrior Assembler for Mac OS

The CodeWarrior assembler for Mac OS is a stand-alone assembler that has an easy-to-use syntax. This is the same syntax used by assemblers for other processor targets supported by CodeWarrior, such as MIPS, ARM, and PowerPC.

For more information about the CodeWarrior assembler, you should read the *Assembler Guide*.

In addition, the C/C++ and Pascal compilers support inline assembly for Mac OS development. See "Inline PowerPC Assembler" on page 229 or "Inline 68K Assembler" on page 339.

CodeWarrior Linker for Mac OS

Like the compilers, the CodeWarrior Mac OS linkers are plug-in tools integrated into the CodeWarrior IDE. They link multiple object modules and libraries, resolving references from one to another to produce executable applications, shared libraries, static libraries, or code resources.

For more information about the linkers, see "PPC Linker" on page 195 and "68K Linker" on page 291.

CodeWarrior Debugger for Mac OS

The CodeWarrior debugger controls your program's execution and allows you to see what's happening internally as your program runs. You use the debugger to find problems in your program's execution.

The debugger can execute your program one statement at a time, and suspend execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, and inspect the contents of the processor's registers.

For general information about the debugger, including all of its common features and its visual interface, you should read the *Debugger User Guide.*

For more information about debugging software for Mac OS, see "Debugging for Mac OS" on page 359.

Metrowerks Standard Libraries

The Metrowerks Standard Libraries (MSL) are standard C and C++ libraries for use in developing applications for Mac OS. The libraries are ANSI compliant, and all of the source for the libraries is provided for you to use in your projects. These are the same libraries that are used for all CodeWarrior targets, but they have been customized and the runtime has been adapted for use in Mac OS development.

For more information about MSL, you should read the *MSL C Reference* and the *MSL C++ Reference*. See also "Mac OS Support Files" on page 391 and "Library Naming Conventions" on page 421.

The Development Process with CodeWarrior

While working with CodeWarrior, you will proceed through the development stages familiar to all programmers: write code, compile and link, debug. For complete information on performing tasks like editing, compiling, and linking, refer to the *IDE User Guide*. For debugging, see the *Debugger User Guide*.

The difference between CodeWarrior and traditional command line environments is in how the software (in this case the IDE) helps you manage your work more efficiently. If you are unfamiliar with an integrated environment in general, or with CodeWarrior in particular, you may find the topics in this section helpful. Each topic discusses how one component of the CodeWarrior tools relates to a traditional command-line environment.

Read these topics to find out how using the CodeWarrior IDE differs from command-line programming.

- Makefiles—the IDE uses a project to control source file dependencies and the settings for compilers and linkers
- Editing Code—an overview of source code editing in the IDE
- Compiling—how the IDE performs compile operations
- Linking—how the IDE performs linking operations
- Debugging—how to debug a program from the IDE
- Viewing Preprocessor Output—A tip on debugging preprocessor directives

Makefiles

The CodeWarrior IDE *project* is analogous to a makefile. Because you can have multiple builds in the same project, in fact the project is analogous to a collection of makefiles. For example, you can have one project that has both a debug version and a release version of your code. You can build one or the other, or both as you wish. In CodeWarrior, these different builds within a single project are called "targets."

The IDE uses the project manager window to list all the files in the project. Among the kinds of files in a project are source code files and libraries.

You can add or remove files easily. You can assign files to one or more different targets within the project, so files common to multiple targets can be managed simply.

The IDE manages all the interdependencies between files automatically, and tracks which files have been changed since the last build. When you rebuild, only those files that have changed are recompiled.

The IDE also stores the settings for compiler and linker options for each target. You can modify these settings using the IDE, or with `#pragma` statements in your code.

Editing Code

The CodeWarrior IDE has an integral text editor designed for programmers. It handles text files in MS-DOS/Windows, UNIX, and Mac OS formats.

To edit a source code file, or any other editable file that is in a project, just double-click the file's name in the project window to open the file.

The editor window has excellent navigational features that allow you to switch between related files, locate any particular function, mark any location within a file, or go to a specific line of code.

Compiling

To compile a source code file, it must be among the files that are part of the current target. If it is, you simply select it in the project window and choose **Compile** from the Project menu.

To compile all the files in the current target that have been modified since they were last compiled, choose **Bring Up To Date** in the Project menu.

In UNIX and other command-line environments, object code compiled from a source code file is stored in a binary file (a ".o" or ".obj" file). The CodeWarrior IDE stores and manages object files transparently.

Linking

Linking object code into a final binary file is easy: use the **Make** command in the Project menu. The **Make** command brings the active project up to date, then links the resulting object code into a final output file.

You control the linker through the IDE. There is no need to specify a list of object files. The project manager tracks all the object files automatically. You can use the project manager to specify link order as well.

Use the **PPC Target** or **68K Target** settings panel to set the name of the final output file.

Debugging

To debug a project, make sure **Enable Debugging** is selected in the Project menu.

For more information on debugging, refer to the *Debugger User Guide*. For information specific to Mac OS debugging, see "Debugging for Mac OS" on page 359.

Viewing Preprocessor Output

To view preprocessor output, select the file in the project window and choose **Preprocess** from the Project menu. A new window appears that shows you what your preprocessed file looks like. You can use this feature to track down bugs caused by macro expansion or other subtleties of the preprocessor.

3

Creating Application Projects for Mac OS

This section shows you how to create new application projects for the Mac OS. The procedure is nearly the same regardless of what kind of project you want, the programming language used, or the host platform on which the code is developed. The only differences between the different kinds of projects are the libraries you use, the project settings, and the target type you are developing for.

TIP: If you find you are creating similar projects all the time and are continuously adding or removing specific libraries, you can create your own project stationery. Refer to the *IDE User Guide* for more detailed information.

Topics discussed in this section are:

- Types of Applications
- Creating a New Application Project
- Creating Fat Applications
- Choosing Libraries for Applications
- Setting Target Options

Types of Applications

CodeWarrior can be used to create four basic types of applications. This section describes these application types:

- 68K
- CFM68K
- PowerPC

- Fat

68K

Applications written for 68K-processor machines run on any Mac OS computer. The code is contained in the resource fork of the application file in resources of type 'CODE'. A Mac OS computer that uses a PowerPC processor contains a software emulator that translates the 68K application instructions into PowerPC instructions. A 68K Mac OS computer runs your 68K code without any translation.

CFM68K

Some applications use the Code Fragment Manager for 68K. This software is contained in a software extension called `CFM-68K Runtime Enabler`, that only works on Mac OS computer with MC68020 or a better chip. It contains its code in a single 'rseg' resource. It's useful if you want the advantages of shared libraries on a 68K Mac OS computer. You must have the `CFM-68K Runtime Enabler` installed on the 68K computer that you want to run your application on.

PowerPC

Applications that are compiled into PowerPC code can run only on a PowerPC-based Mac OS. A PowerPC application contains its code in the data fork of the file. The resulting application is almost always faster than a 68K application, since PowerPC processors are very fast, and because no translation of instructions is required.

Fat

Fat applications run on a 68K or on a PowerPC-based Mac OS computer. A Fat application contains both 68K or CFM68K code and PowerPC code (in the data fork). It runs without translation on both 68K and PowerPC computers, but the application's code size is larger than either 68K or PowerPC applications since both types of instructions are compiled into the program.

Creating a New Application Project

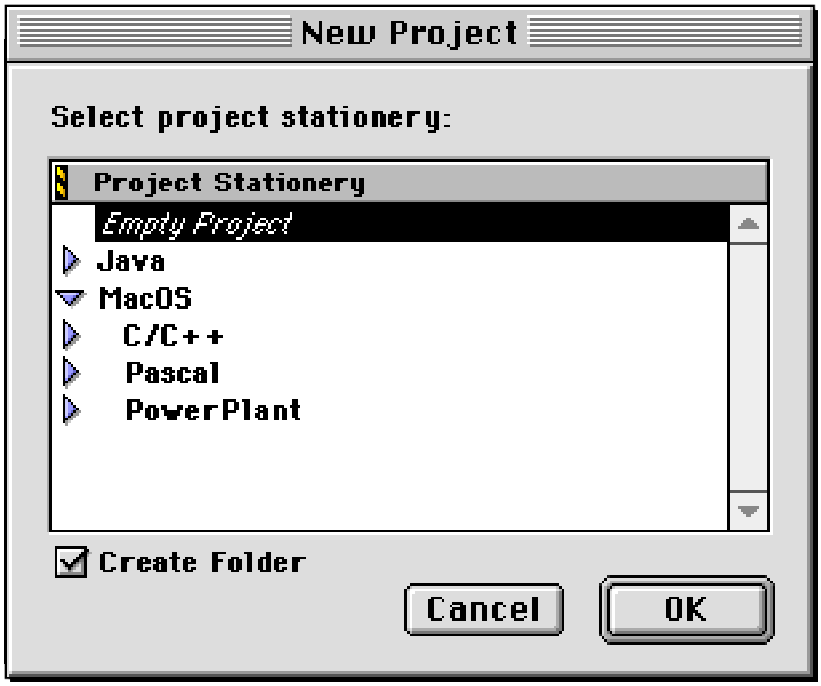
Except for Fat projects, all projects should be created from Project Stationery. This reduces the need to add the standard libraries to a project manually. You should then only need to add extra libraries if your particular code requires them.

To learn how to create a Fat application, refer to "Creating Fat Applications" on page 39.

1. **Create a new project.**

   To create a new project, choose **New Project** from the **File** menu. The New Project dialog box is displayed, as shown in Figure 3.1.

Figure 3.1 New Project window



2. **Choose a stationery project.**

   Click the disclosure triangles to the left of the Project Stationery text items, shown in Figure 3.1, to expand or collapse the list of project stationary available to you. Choose the appropriate stationery

project for your language and target project. Table 3.1 shows a comprehensive list of the Mac OS stationery projects available.

Table 3.1 Choosing project stationery

Language Category	Available Stationery
C/C++	MacOS Toolbox 68K MacOS Toolbox Multi-Target MacOS Toolbox PPC Std C Console 68K Std C Console CFM68k (DLL) Std C Console Multi-Target Std C Console PPC Std C Console PPC (DLL) Std C++ Console 68K Std C++ Console CFM68K (DLL) Std C++ Console Multi-Target Std C++ Console PPC Std C++ Console PPC (DLL) SIOUX-WASTE C 68K SIOUX-WASTE C Multi-Target SIOUX-WASTE C PPC SIOUX-WASTE C++ 68K SIOUX-WASTE C++ Multi-Target SIOUX-WASTE C++ PPC

Language Category	Available Stationery
Pascal	ANS Console 68k ANS Console Multi-Target ANS Console PPC Basic Toolbox 68k Basic Toolbox Multi-Target Basic Toolbox PPC Object Pascal 68k Object Pascal Multi-Target Object Pascal PPC
PowerPlant	PowerPlant Appearance 68K PowerPlant Appearance FAT PowerPlant Appearance PPC PowerPlant Basic 68K PowerPlant Basic FAT PowerPlant Basic PPC PowerPlant Documents 68K PowerPlant Documents FAT PowerPlant Documents PPC PowerPlant Net Client 68k PowerPlant Net Client FAT PowerPlant Net Client PPC PowerPlant Net Server 68k PowerPlant Net Server FAT PowerPlant Net Server PPC

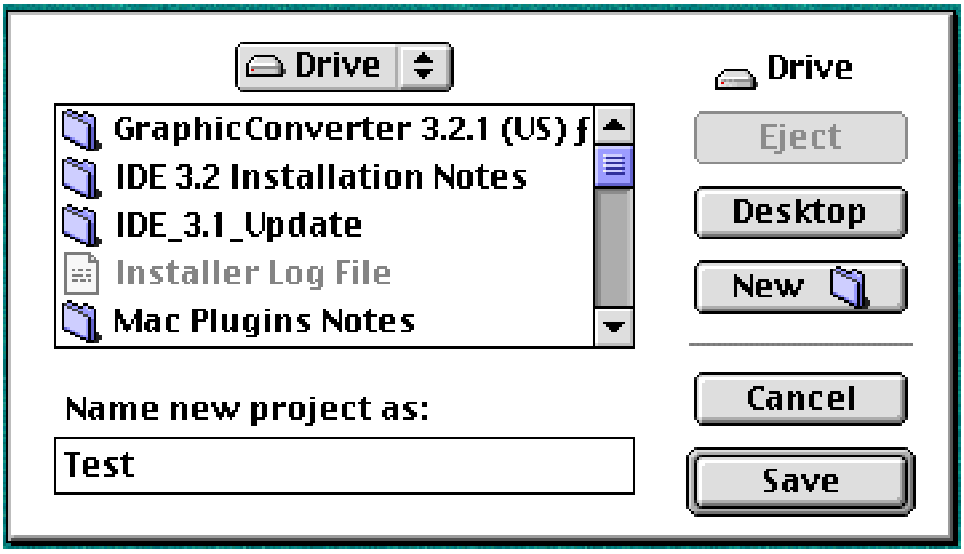
Default settings are used for all options in these stationery project files. If you want to change the defaults, you can do so by creating your own project stationery. To learn how to do this, refer to the *IDE User Guide* for more detailed information. You'll find this manual on the CodeWarrior CD.

NOTE: The exact names of the stationery files could be slightly different, and more stationery than is listed here may be available to you.

3. **Name the project.**

   Enter a name for the project and click **Save** in the dialog box that appears, as shown in Figure 3.2.

Figure 3.2 Name Your Project



After you name your project and click Save, a new project window appears on your screen, as shown in Figure 3.3.

4. **Add any required extra libraries.**

   Refer to "Choosing Libraries for Applications" on page 62 for more information on additional libraries you may need to add to your project.

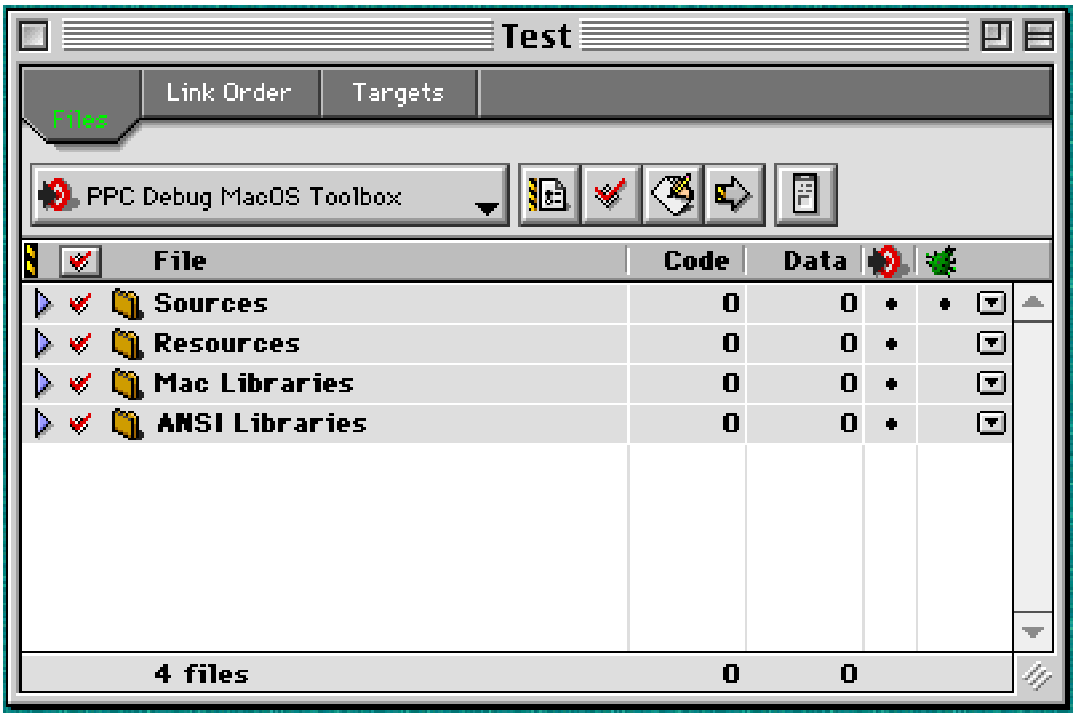
   If you don't know how to add files to a project, refer to the *IDE User Guide* on the CodeWarrior CD.

5. **Configure your project options.**

   Now that you have a new project started, you can configure the Preferences and Target Settings for the project using these menu commands on the Edit Menu.

   For more information on setting project options, refer to "Setting Target Options" on page 68.

Figure 3.3 New Project Window



6. **Build your project, debug it, and sell one million copies.**

   To learn how to build your project, refer to the *IDE User Guide* documentation on the CodeWarrior CD.

   To learn about debugging, you can refer to the *Debugger User Guide* documentation on the CodeWarrior CD, and also refer to "Debugging for Mac OS" on page 359.

Creating Fat Applications

This section discusses how to create a Fat application. This procedure has changed recently due to changes in the CodeWarrior IDE.

A Fat application is one that contains code for both 68K and PowerPC-based Mac OS computers. A Fat application will run natively on either system. In other words, on PowerPC Mac OS computers, the PowerPC instruction will be executed. On 68K Mac OS computers, then 68K instructions will be executed.

The following discussion describes two completely different, yet elegant ways to create a Mac OS Fat application. You can read more about the issues surrounding these strategies in the *IDE User Guide*. You will find the discussion there under the topic of Complex Projects. There are several ways to accomplish this task. By reviewing all the methods, you can get a good overview of possibilities using the latest features of the CodeWarrior IDE.

Which method is better for you? It depends on whether you like the build-process modularity of subprojects. Multiple targets is a nice way to go if you want to just have one project file. Using multiple targets, you can have both debug and shipping versions of both 68K and PowerPC, created as targets within one project. If it isn't obvious to you which one you would prefer, read the descriptions and decide for yourself.

The method using subprojects is perhaps easier to accomplish for novices, though the trade-off is that slightly more build time is incurred due to the overhead of checking the build dependencies for the subproject. The multiple target approach uses the full sophistication of the CodeWarrior IDE's features. Also, the multi-target approach will make it easier in the future to keep the project's file lists more manageable.

The methods discussed for creating Fat applications are:

- Creating Fat Applications with Multiple Targets
- Creating Fat Applications with Subprojects

Creating Fat Applications with Multiple Targets

This method leads you through the process of building a Fat application by creating multiple targets in one project file. This example is slightly complex, involving many steps. However, when you create a Fat application using this method, you will see many of the advanced features of the CodeWarrior IDE put to work.

1. **Create a 68K application projects: a 68K or CFM68K project and a Power Macintosh project.**

   Before you can create a Fat application, you must create working 68K and Power Macintosh applications. To do this, create two dif-

ferent projects by performing the procedure outlined in "Creating a New Application Project" on page 35 two separate times. Save both projects in the same folder by de-selecting the **Create Folder** checkbox on the **New Project** dialog. If the **Create Folder** checkbox is selected, a new folder is created for each project you generate.

You may be tempted to bypass this step somehow. Don't do it! Take the time to make sure you have two properly-functioning applications built from two different projects, and you will avoid trouble in the rest of this procedure.

WARNING! Create each of the two projects as outlined in the procedure discussed above, preferably using the relevant project stationery. Don't try to duplicate one project and convert it. If you try this anyway against our advice, be advised that extravagant effects may be observed.

Include the same source code files and resource files in each project file. Only the library files should be different, since 68K and PowerPC applications need to link with different libraries.

If you need to use different source code for the PowerPC and 68K applications, use the directive `#if defined(powerc)` to bracket code that depends on the processor you're compiling for. For example,

```
#if defined(powerc)
    /* do nothing for PowerPC */
#else
    SetA5(oldA5); /* set A5 world for 68K */
#endif
```

2. **Test and debug the PowerPC and 68K projects.**

   Both applications must work correctly before you can create a Fat application. To learn how to build your projects, refer to the *IDE User Guide*. To learn how to operate the debugger, refer to the *Debugger User Guide*.

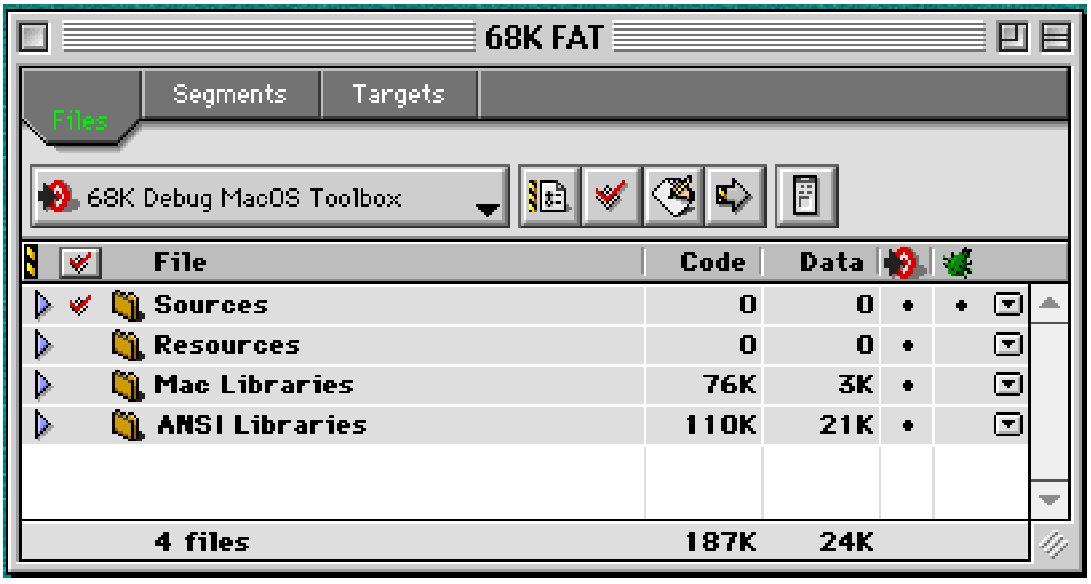
3. **Make backup copies of both your projects.**

   You should save a copy of each of your projects in case you make an error and want to start over again. You can use the **Save A Copy As** menu command in the File Menu to save a backup copy of each project.

4. **Open your 68K project.**

   Open one of your projects (we'll use the 68K project here). Your project should look something like Figure 3.4. Note that the Code and Data columns have numbers in them, meaning that the source code files have been compiled. If you haven't built your project recently, do it now to make sure it builds.

Figure 3.4 A 68K Project Window



5. **Create a New Target.**

   Click on the Targets category tab in the Project window shown in Figure 3.4. The view switches to the Targets view. Now use the Projects Menu to invoke the Create New Target command. Give your target a name using the Name For New Target editable text field, as shown in Figure 3.5. The name should be something that signifies a PowerPC target name. Then, click on the **Clone Existing**

Target radio button. This makes the new target a copy of the existing target selected with the drop-down menu. Since there is only one target in our example file, this is the one that will be cloned.

Figure 3.5 Creating a New Target

Now that you have created a new target, the Targets view of the Project window should look similar to that shown in Figure 3.6. Note that at this point, the project will not build correctly. We have some more work to do on it.

6. **Configure the PowerPC target with PowerPC settings.**

   To finish making your PowerPC target, you need to assign PowerPC attributes to it. To do this, click on the Targets tab in the Project window, so that the targets are showing, as shown in Figure 3.6. Then, double-click on the PowerPC target. You will see a dialog box similar to that shown in Figure 3.7. Click on the **Target Settings** item in the Target Settings Panels list on the left side of the dialog box. The view changes to that shown in Figure 3.8.

   Choose **Mac OS PPC Linker** from the **Linker** pop-up menu, as shown in Figure 3.8. You will then see a dialog box as shown in Figure 3.9. Go ahead and click OK to dismiss the dialog box.

Figure 3.6 Multiple Targets in the Project Window

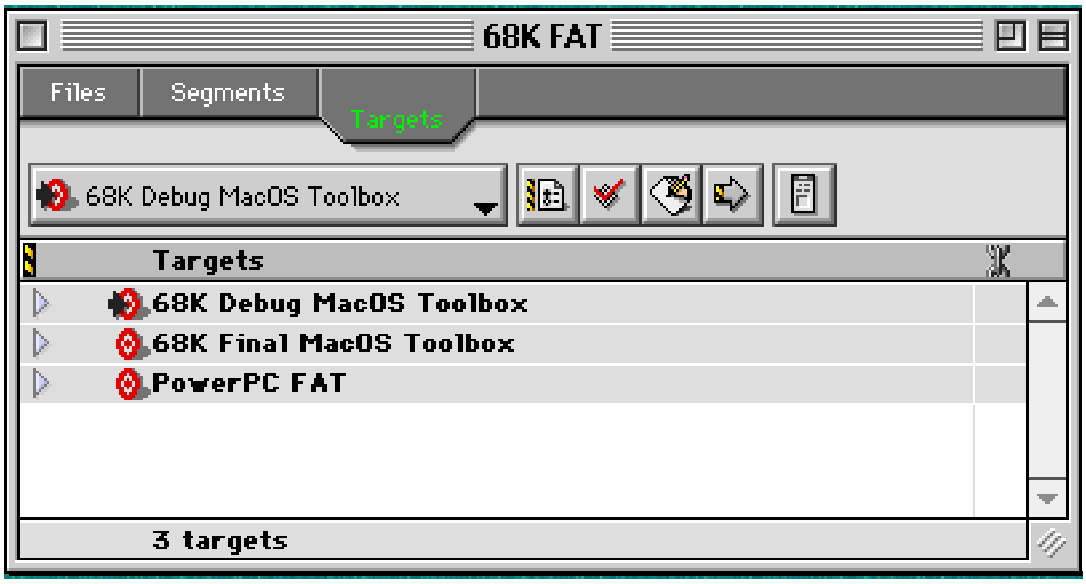


Figure 3.7 Target Settings Dialog Box

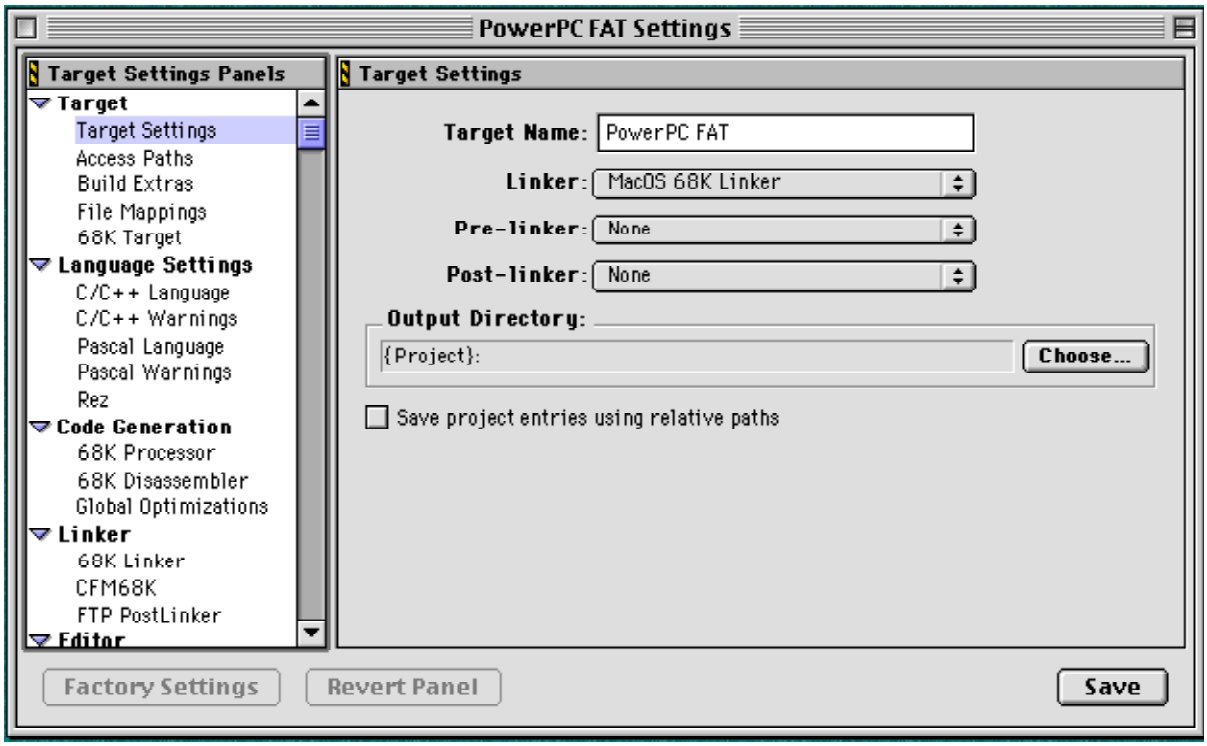


Figure 3.8 Target Settings Panel

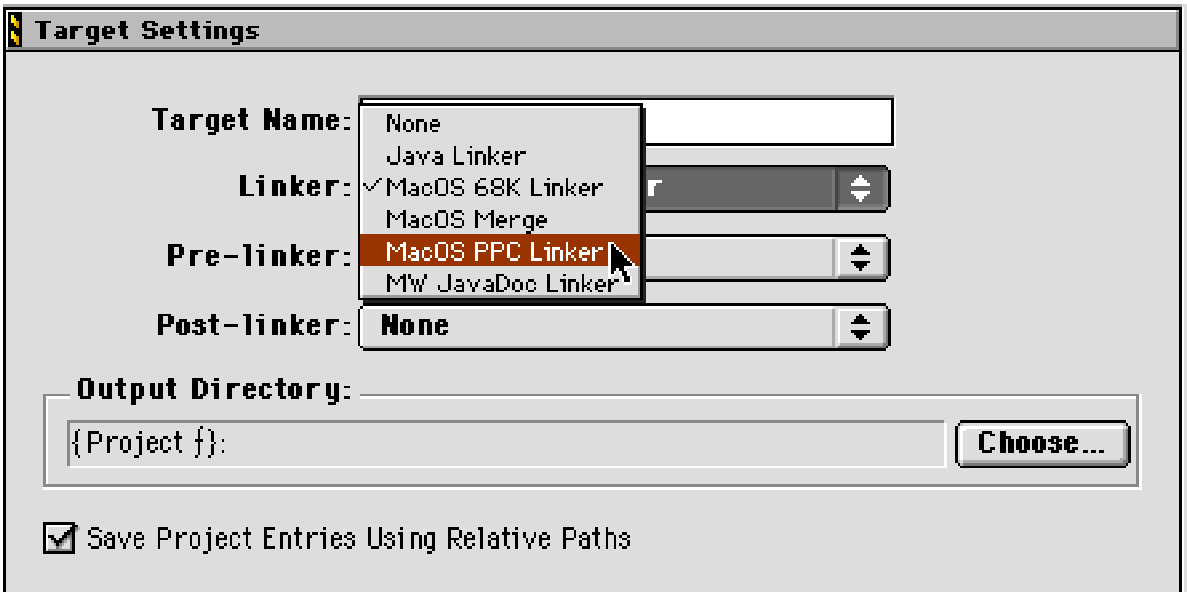
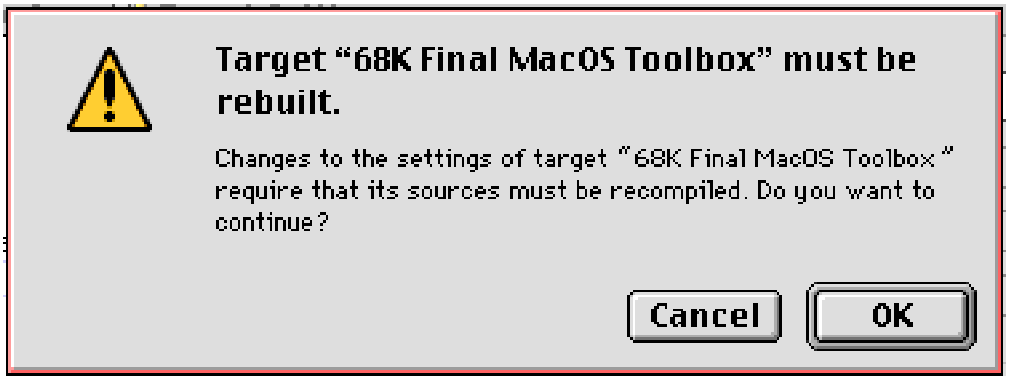
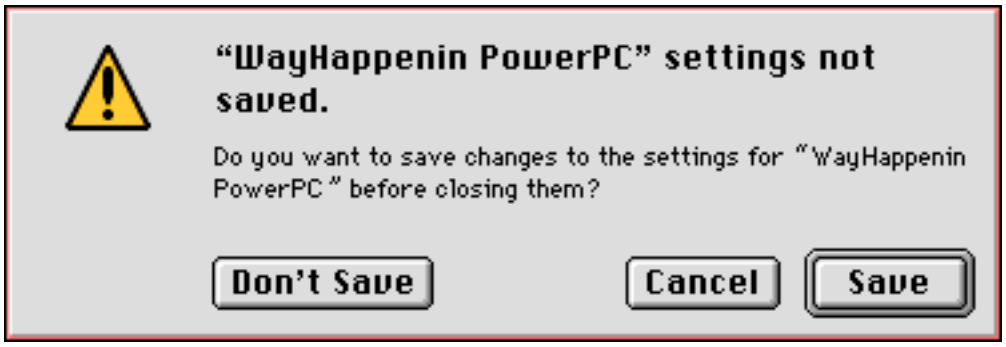


Figure 3.9 Project Must Be Rebuilt Dialog Box


Target "68K Final MacOS Toolbox" must be rebuilt.
Changes to the settings of target "68K Final MacOS Toolbox" require that its sources must be recompiled. Do you want to continue?
Cancel
OK

Now you can close the Target Settings dialog box. When you do this, another dialog box will be presented, as shown in Figure 3.10, Click **Save** to close it.

Figure 3.10 Save Settings Dialog Box


"WayHappenin PowerPC" settings not saved.
Do you want to save changes to the settings for "WayHappenin PowerPC" before closing them?
Don't Save
Cancel
Save

7. **Assign files to the correct targets.**

   Now what is needed is to assign all the files in the Files view of the Project window to the correct targets. When you created a clone of your existing target in the last step, all the library files that were in the 68K project were assigned to also be in the PowerPC project. At link time, the 68K libraries will give errors when building the PowerPC target, so this needs to be fixed. To do this, we'll remove all the 68K library files from the PowerPC target you just created.

   Switch to the Files view of the Project window, by clicking on the Files tab. Now, choose the Project Inspector menu command from the Window men. Then click on the file name of one of the files in the Files view of your Project window. The Project Inspector window should look something like that shown in Figure 3.11. This window is giving you information about the files in your Project window Files view. As you click on different files in the Files view of your Project window, the Project Inspector window changes to give you new information about the file you clicked on. For detailed information on the Project Inspector window, refer to the *IDE User Guide* on your CodeWarrior CD.

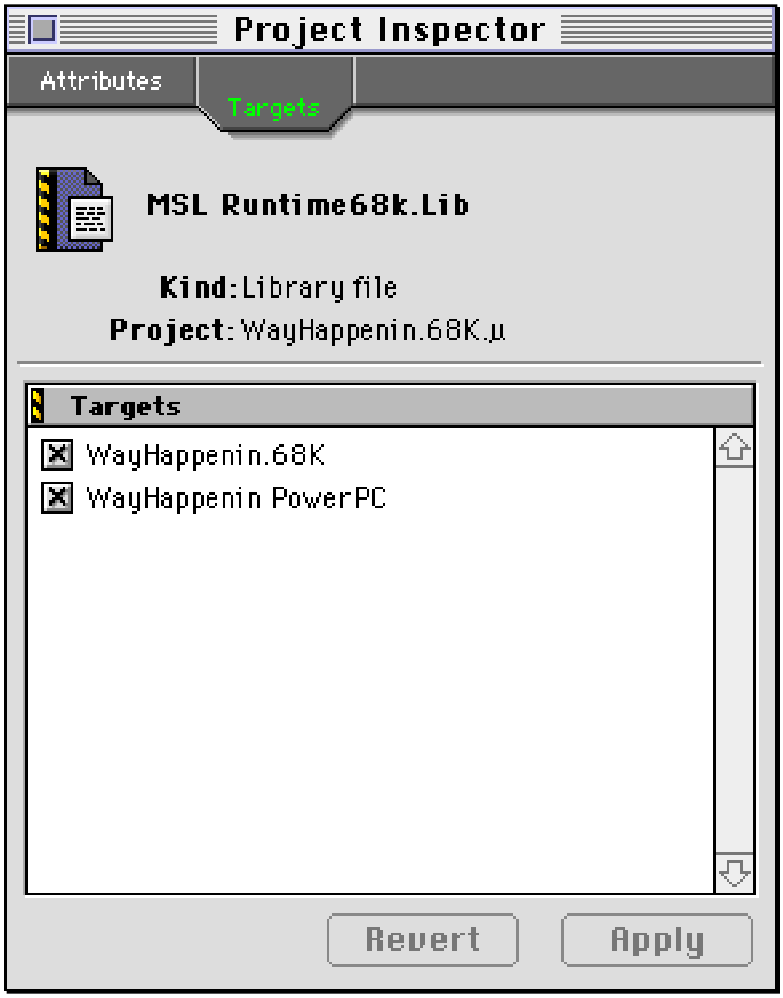
Figure 3.11 Project Inspector Window

Click on one of the library files in the 68K project. Then click on the Targets tab of the Project Inspector window. You should see something like the window shown in Figure 3.12. This window shows that the 68K library file is assigned to both the 68K and PowerPC targets, and this is incorrect for our purposes. To change this, click on the check box for the PowerPC target so that it is not checked. By doing this, we remove this library file from the PowerPC target. We will need to add the PowerPC libraries in a later step.

Keep going through the Files view of the Project window, and for every library file that is for 68K projects, remove the file from the

PowerPC target using the Targets view of the Project Inspector Window.

Figure 3.12 Targets Tab of Project Inspector Window



8. **Add the PowerPC library files to the project.**

   Now that you have removed all the 68K-specific libraries from the PowerPC target, you need to add the required PowerPC library files to the project.

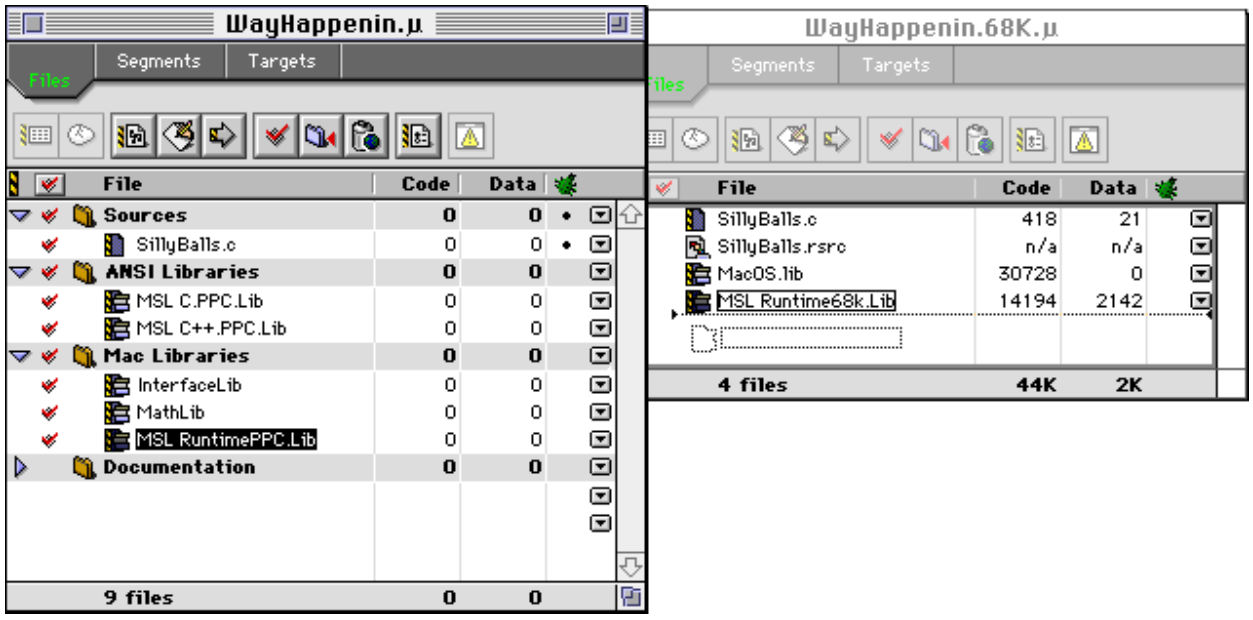
   There are two ways to do this. You can use drag and drop, or you can choose **Add Files** from the **Project** menu. We'll use drag and drop here so you can see how easy it makes this operation.

   First, make sure that you have both of the needed project files open. You should open the PowerPC version of your project (the one that you got working in steps 1 and 2 of this procedure) and the project file you have been working with. What you're going to do is drag and drop files from the PowerPC project to the new project, and assign the library files to the PowerPC target as you drag the files.

   To simplify this step, select all the library files in the PowerPC project that you will need to move, since you can drag and drop multiple files at a time. Optionally, you can move one file at a time, but that requires redoing this step multiple times. If you don't know how to select multiple files in the Project window, refer to the *IDE User Guide* to learn how.

   Let's move one file for this example, just to keep the illustration simple. Click on one of the PowerPC libraries in the PowerPC project, then drag the selected file over to the new project window. This drag and drop operation is shown in progress in Figure 3.13.

Figure 3.13 Adding a File to Your Project Using Drag and Drop



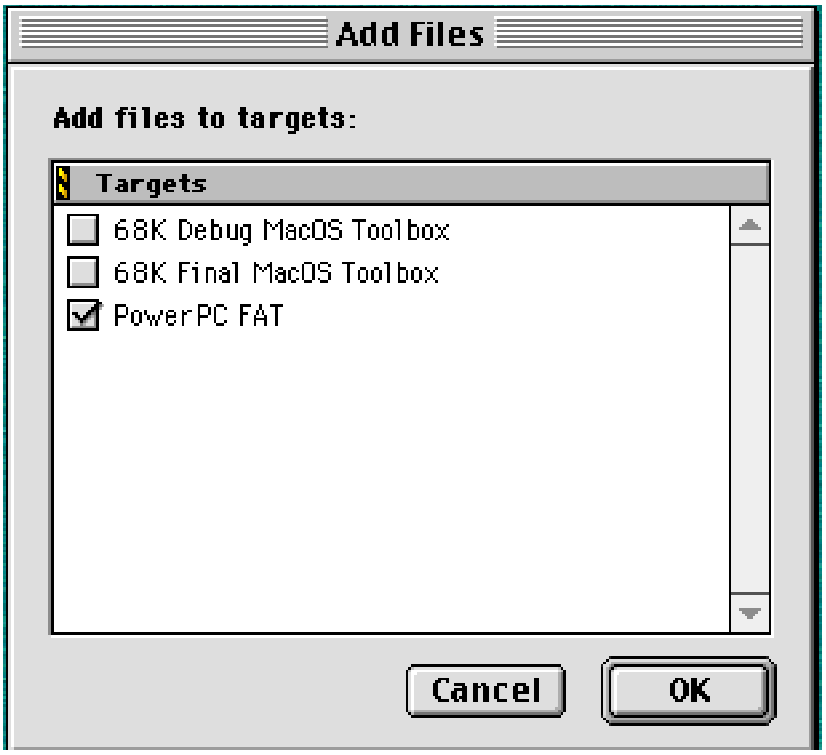
When you let go of the file after dragging it to the new project, a window similar to that shown in Figure 3.14 appears. Since the library you are adding to your project is only for the PowerPC target, you should make sure that the check box for the 68K target is not checked, as shown in Figure 3.14. This figure shows that the file you are adding to the project will go in the build for the PowerPC target, but will not be used for the 68K build target.

When you are done adding the file to the project, your project window will resemble Figure 3.15.

Repeat this step until you have all the PowerPC libraries added to the PowerPC target of your new project. If you want, you can group all the libraries using the grouping feature of the Project window. Refer to the *IDE User Guide* to learn how to do this.

You can use the Project Inspector window, or the Segments or Link Order views of the project window in the IDE to see if the libraries you add are assigned to the intended targets.

Figure 3.14 Assigning the Target for Added Files



9. **Rename your project file.**

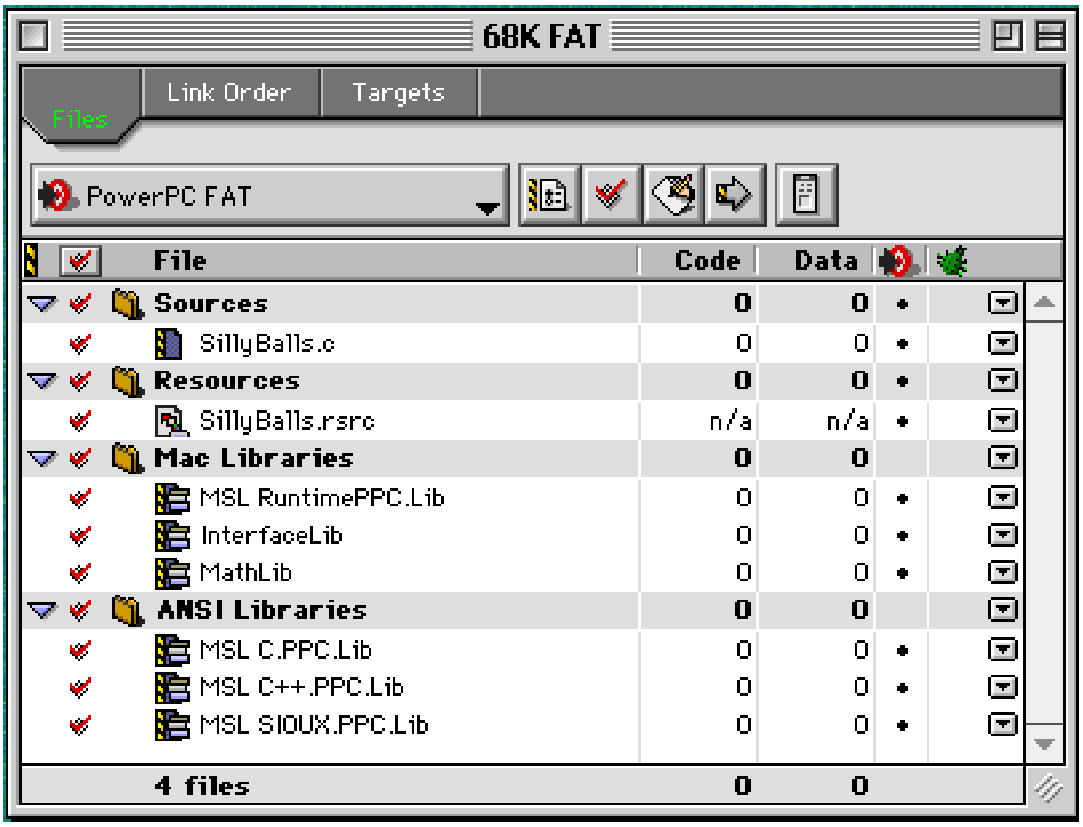
   Since your original project file may not be named appropriately, you should take the time to rename it now, using the **Save A Copy As** command on the **File** menu. Name it something that tells you it's a Fat project.

10. **Build your PowerPC project.**

    Use the **Set Current Target** menu command on the **Project** menu to select your PowerPC target. Then use the **Make** command to build it. If you get any link errors, it is probably due to having libraries assigned to the wrong targets. Use the Project Inspector window to correct any of these errors that occur.

    If you want, you can use **Set Current Target** and **Make** again on your 68K target to make sure that it still builds.

Figure 3.15 Project Window with PowerPC Libraries Added



11. Create a Fat Target.

Now that you have a 68K and a PowerPC target in your project file, you need to create one more target to bring them together at link time. To do this, go to the Targets view of the Project window. Then use the **Create New Target** command on the **Project** menu to create a new target. The dialog box shown in Figure 3.16 appears.

Figure 3.16 Create New Target Dialog Box

Use the **Name For New Target** editable text field to give your new target a name, then click **OK**. This time, the **Empty Target** radio button will suit our purposes, and does not require us to change it. Then, double-click on the new target in the Targets view of the Project window, and click on **Target Settings** from the list on the left.

Choose **Mac OS Merge** from the **Linker** pop-up menu (Figure 3.17). Then click the **OK** button in the dialog box that appears. Then, click the close box for the window and click on the **Save** button in the dialog box that appears.

12. **Name the merged output file.**

    Double-click on the Fat target in the Target view of the Project window to bring up the Target Settings dialog. Then, click on **Mac OS Merge** in the Target Settings Panels list box on the left side of the dialog box. You should see the right side of the dialog box change to the view shown in Figure 3.18. Use the File Name editable text field to give your final Fat application file a name. Then click the Save button and close the dialog box, saving any changes you made by clicking Save in the dialog box that appears.

Figure 3.17 Selecting the Mac OS Merge Linker

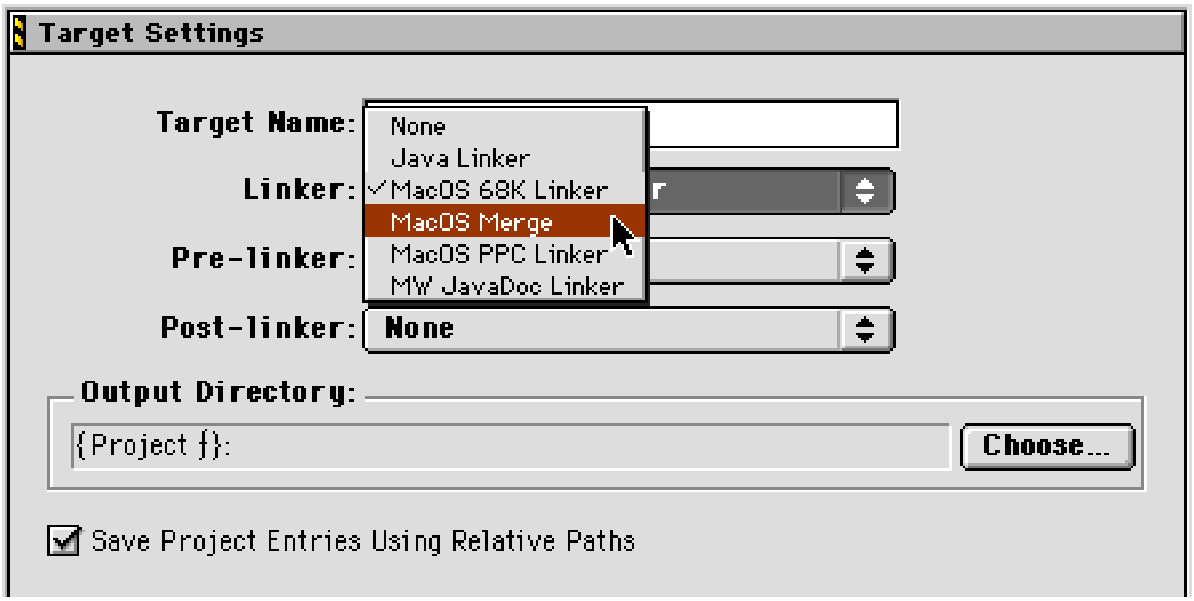
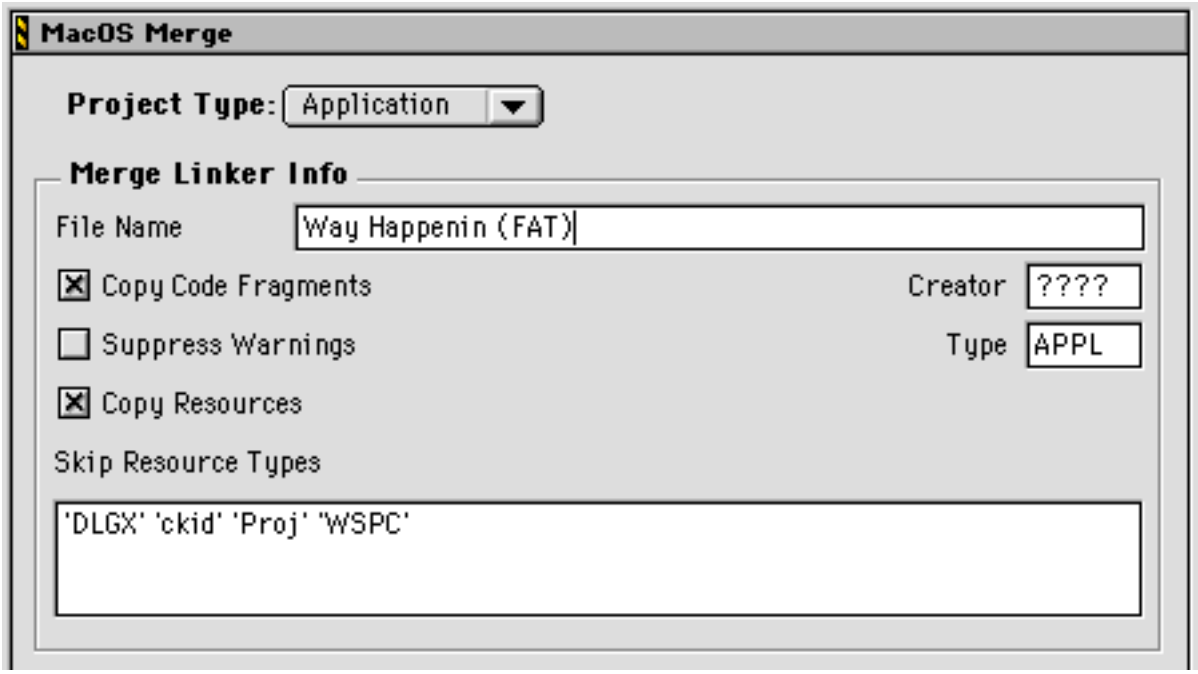


Figure 3.18 Mac OS Merge Panel



13. **Create dependencies between the Fat target and the other targets.**

    In the Targets view of the Project window, click on the 68K targets, and drag and drop them onto the Fat target. Do the same thing with the PowerPC target. Then, click the disclosure triangle on the left side of the Fat target, to show that the targets are now dependents of the Fat target, as shown in Figure 3.19. This means building the Fat target will create the 68K and PowerPC targets.

    Now click in the Link Column on the far right side of the window, next to the targets you dragged onto the Fat target. You should see a black dot appear as you click once in the column for each target, as shown on the right side of the window in Figure 3.19. This tells the IDE to use the output file of the target at link time for the target that depends on this. In other words, the Fat target depends on the 68K and PowerPC targets, and so when they are linked, the output files will be used to merge-link with the Fat target.

Creating Fat Applications with Subprojects

This method leads you through the process of building a Fat application by creating two separate project files. After you create the two separate files, you create a dependency relationship between them.

1. **Create two application projects: a 68K or CFM68K project, and a PowerPC (PPC) project.**

   Before you can create a Fat application, you must create working 68K and Power Macintosh applications. To do this, create two different projects, by twice performing the procedure outlined in "Creating a New Application Project" on page 35.

   You may be tempted to bypass this step somehow. Don't do it. Take the time to make sure you have two properly-functioning applications built from two different projects, and you will avoid trouble in the rest of this procedure.

WARNING! Create each of the two projects as outlined in the procedure discussed above, preferably using the relevant project stationery. Don't try to duplicate one project and convert it. If you

try this anyway against our advise, be warned that consequences might be severe.

Figure 3.19 Fat Target with 68K and PowerPC Dependencies

Include the same source code files and resource files in each project file. Only the library files should be different, since 68K and PowerPC applications need to link with different libraries.

If you need to use different source code for the PowerPC and 68K applications, use the directive `#if defined(powerc)` to bracket code that depends on the processor you're compiling for. For example,

```
#if defined(powerc)
    /* do nothing for PowerPC */
#else
```

```
    SetA5(oldA5); /* set A5 world for 68K */
#endif
```

2. **Build, test, and debug the PowerPC and 68K projects.**

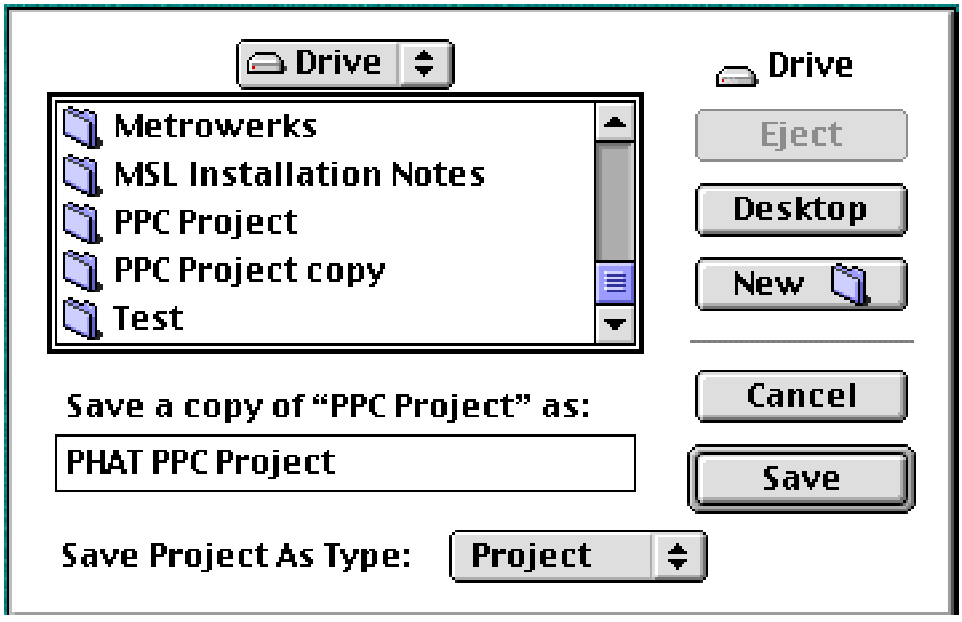
   Both applications must work correctly before you can create a Fat application. To learn how to build your projects, refer to the *IDE User Guide*. To learn how to operate the debugger, refer to the *Debugger User Guide*.

3. **Create a duplicate of your PowerPC Mac OS project.**

   Make a duplicate of the PowerPC application project file. If you are developing your code using a Mac OS computer, use the **Duplicate** command in the Mac OS Finder, or the **Save A Copy As** command in the **File** menu. If you are developing your code on a Windows computer, use the **Save A Copy As** command in the **File** menu.

   Rename the duplicated project to something descriptive, such as word play on "fat". Figure 3.20 shows the dialog box that appears when you choose the **Save A Copy As** menu command.

Figure 3.20 Saving a Duplicate of Your Fat Project File

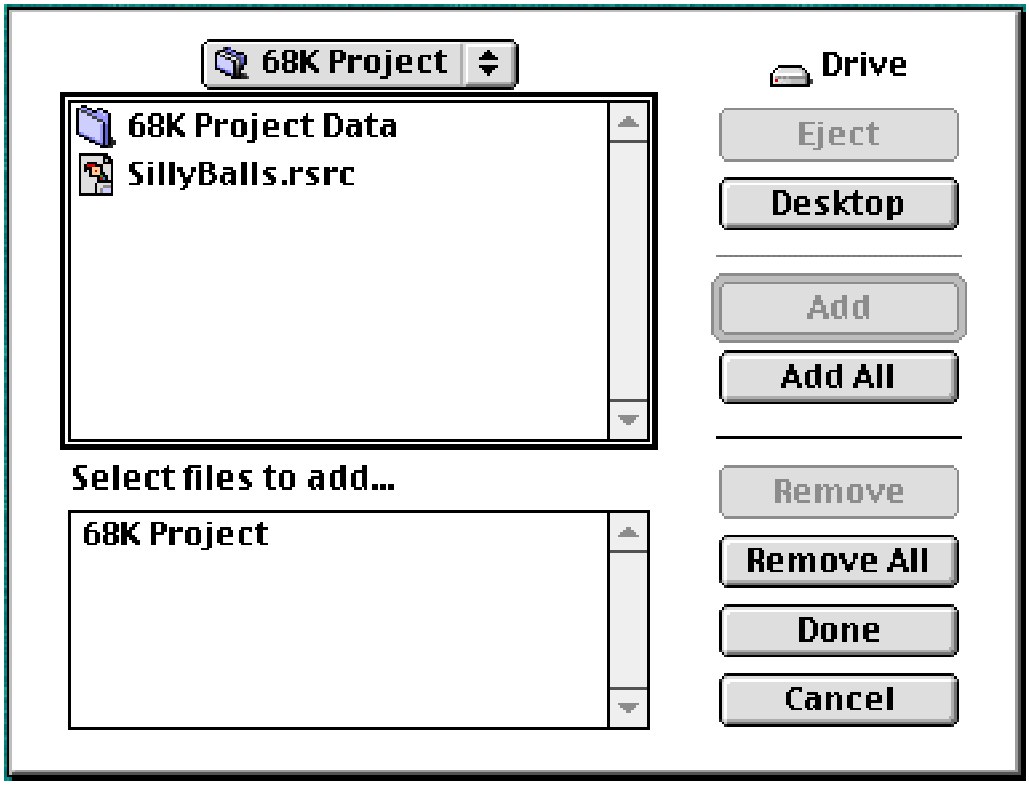


4. **Add the 68K project file (as a subproject) to the Fat application project.**

   Click in your open Fat project file window to make the window active, and then use the **Add Files** command in the **Project** menu to add the 68K project file to the project. This makes the 68K project file a subproject of the Fat project. When the Fat project is built, the 68K project will automatically be updated.

   Figure 3.21 shows the dialog box that appears when you use the Add Files menu command for selecting a file to add. Click on the 68K project file you want to add, then click the Add button. Then click the Done button.

Figure 3.21 Selecting a 68K Project File to Add to the Fat Project



After you add this file to the Fat project, an Add Files dialog appears. Because you only want to add the 68K project to the Fat

project, you should make sure that is the only checkbox selected, as shown in Figure 3.14.

Figure 3.22 Indicating which target to add file to

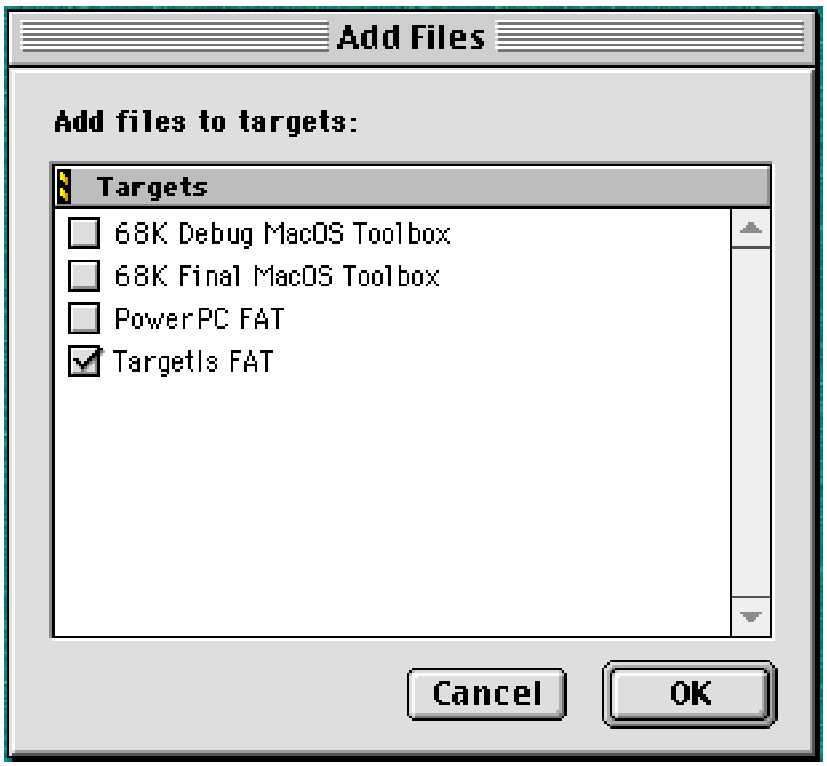


Figure 3.23 shows the Fat project file with the 68K project file added to it.

5. **Remove the resource files from the Fat application.**

   This step is optional. When you do this step, you will minimize the number of warnings you receive at link time for the Fat project.

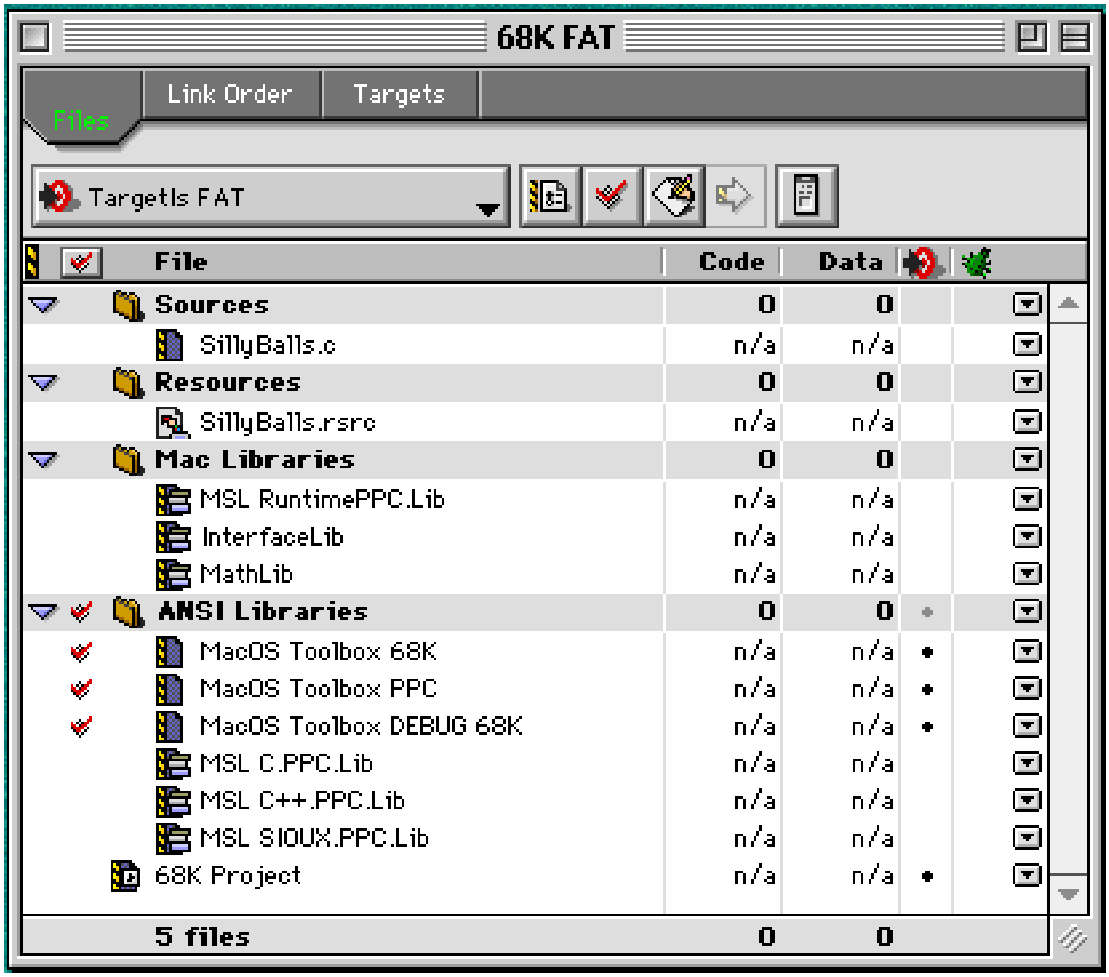
   Remove all resource files from your Fat project file, using the **Remove Selected Items** menu command in the **Project** menu. The reason you do this is to keep duplicate resources from being used at link time, which generates link warnings.

   You aren't required to remove the resource files from the Fat project, this step is optional. But the CodeWarrior IDE creates dupli-

cate resource warnings if you don't. You can safely ignore the warnings if they don't bother you.

If you don't remove the resource files, you should be aware that CodeWarrior will copy the 'SIZE' resource from the 68K application into your final linked Fat application. It will ignore the Preferred Heap Size, Minimum Heap Size, and SIZE Flags settings in the Fat project's **Target Settings** panel. Refer to "Setting an application's 'SIZE' resource" on page 71 for a description of the 'SIZE' resource.

Figure 3.23 Fat Project File with 68K Subproject File



6. **Configure linking for the 68K application.**

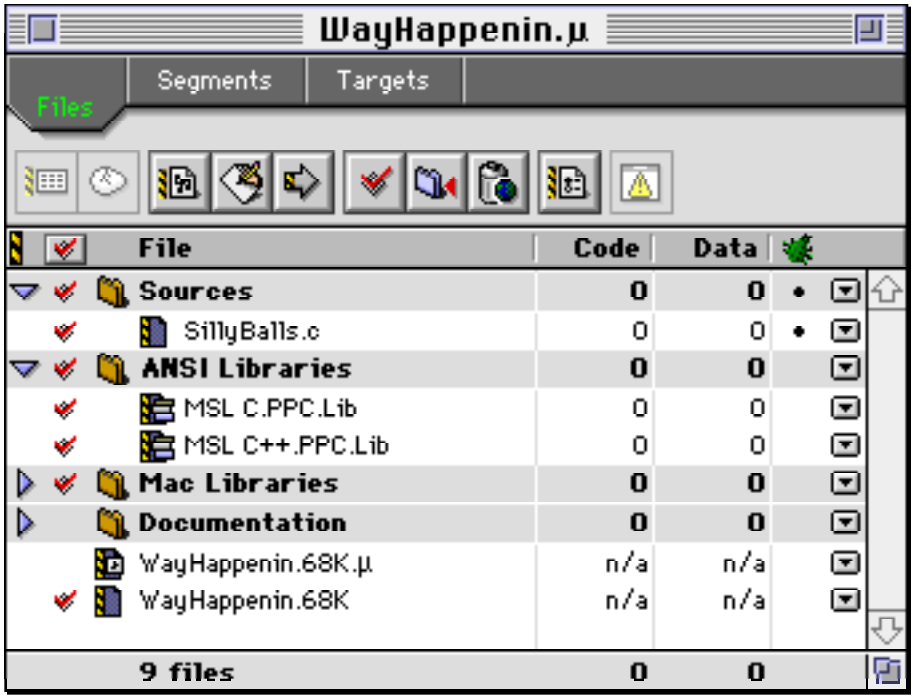
   Click in the Link Column of the Targets view of the Project window, in the same row as the 68K targets, as shown in Figure 3.24. When you click there, small dots appear in the column.

Figure 3.24 Configuring the Linking of 68K Code into the Fat Application

This causes the 68K target output files to be linked into the Fat target's output file. So now, when the Fat target is built, the 68K subproject is built first. Then the output of the 68K target is linked into the Fat target.

Notice that when you click on the Link Column, the Files view of the Project window changes, as shown in Figure 3.25. The output file of the 68K target has now been added to the project as a file.

Figure 3.25 Including the 68K File in the Project Window Files View



7. **Build the Fat application project.**

   Use the **Make** command on the **Project** menu to build and link your Fat project. CodeWarrior creates a new Power PC application and copies the code resources, and the project resources, from the 68K application into the Fat application.

Choosing Libraries for Applications

This section describes which CodeWarrior libraries you should include in your Mac OS application projects.

Generally, you'll create an application project with project stationery, so you won't need to add libraries on your own. But if you change certain project options, this section explains which libraries you may need to change.

You might find it useful to understand the naming convention for libraries. To read about this, refer to "Library Naming Conventions" on page 421.

The topics in this section include:

- A Note About SIOUX
- 68K C/C++ Libraries
- Power Macintosh C/C++ Libraries
- CFM68K C/C++ Libraries
- 68K Pascal Libraries
- Power Macintosh Pascal Libraries

A Note About SIOUX

The text in this section of the manual may mention SIOUX console windows. A SIOUX console window is much like a dumb terminal or TTY. You can write to it and read from it with the standard C functions and C++ operators, such as `printf()`, `scanf()`, `getchar()`, `putchar()`, <<, and >>. It helps you create programs with a simple interface or port programs with a command line interface (CLI) such as DOS and UNIX programs.

If you use an MSL C or C++ library, you should know that these libraries contain Input/Output (IO) routines, such as `printf`(), and `cout`. This means that you will get linker errors if you don't have a SIOUX library in your project.

You use the file `console.stubs.c` in lieu of using SIOUX libraries when you want to use Standard C Library functions such as string routines or memory routines, but do not need to use the SIOUX input and output window.

`The console.stubs.c` file suppresses the initialization overhead of the console windows by using stub (do nothing) functions, minimizing the size of your code. The console_io source files are used to build the libraries, and you would not normally want to add these files to your project.

68K C/C++ Libraries

Table 3.2 lists all the CodeWarrior libraries you might need to add. to a 68K C or C++ project.

Table 3.2 Choosing libraries for a 68K application project

If the project…	Add this…
Calls any Macintosh Toolbox routines	`MacOS.lib`
Contains any C++ code	`MSL Runtime68K.Lib`
Uses custom startup code	MSL CustomAppl68KStartup.Lib
Calls any math routines	`MathLib68K` *xx*`..Lib`
Calls any ANSI C standard library routines	`MSL C` *xx*`.Lib` libraries, such as `MSL C.68K (2i).Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Uses Standard C Library functions such as string routines or memory routines, but doesn't need to use the SIOUX input and output window.	`console.stubs.c`
Calls any SIOUX functions	`MSL SIOUX.68K.Lib`
Contains any C++ code and calls any ANSI C standard library routines	`MSL C++`*xx*`.Lib` libraries, like `MSL C++.68K.Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.

The MSL SIOUX library enables you to write applications that use a standard console window. If you are using SIOUX, the `MSL SIOUX.68K.Lib` file must be added to your project, in addition to any other libraries you require. However, if you wish to write your own console window, remove the SIOUX source code and use `console.stubs.c`. This file provides the required stub procedures for you to write as a library.

If the library's name in the table contains *xx*, you must choose a library based on some settings panel options. To learn how to choose a library by name, refer to "Library Naming Conventions" on page 421 to read about the naming conventions for libraries.

Power Macintosh C/C++ Libraries

Table 3.3 lists all the CodeWarrior libraries you might need to add.

Table 3.3 Choosing libraries for a Power Macintosh application project

If the project…	Add this…
Calls any Macintosh Toolbox routines	`InterfaceLib`
Calls any math routines	`Mathlib`
Calls any ANSI C standard library routines and uses the SIOUX console window	`MSL C.PPC.Lib`
Contains any C++ code and calls any ANSI C standard library routines	`MSL C++.PPC.Lib`
Uses Standard C Library functions such as string routines or memory routines, but doesn't need to use the SIOUX input and output window.	`console.stubs.c`
Calls any SIOUX functions	`MSL SIOUX.PPC.Lib`

The MSL SIOUX library enables you to write applications that use a standard console window. If you are using SIOUX, the `MSL SIOUX.PPC.Lib` file must be added to your project, in addition to any other libraries you require. However, if you wish to write your own console window, remove the SIOUX source code and use `console.stubs.c`. This file provides the required stub procedures for you to write as a library.

If the library's name in the table contains *xx*, you must choose a library based on some settings panel options. To learn how to choose a library by name, refer to "Library Naming Conventions" on page 421 to read about the naming conventions for libraries.

CFM68K C/C++ Libraries

Table 3.4 lists all the CodeWarrior libraries you might need to add to Code Fragment Manager applications.

Table 3.4 Choosing libraries for a CFM68K application project

If the project…	Add this…
Calls any Macintosh Toolbox routines	`InterfaceLib`
Calls any math routines	`MathlibCFM68K (4i/8d).Lib`
Calls any ANSI C standard library routines	`MSL C` *`xx`*`.Lib` libraries, such as `MSL C.68K (4i).Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Contains any C++ code and calls any ANSI C standard library routines	`MSL C++` *`xx`*`.Lib` libraries, like `MSL C++.68K (4i).Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Calls any SIOUX functions	`MSL SIOUX.68K.Lib`

68K Pascal Libraries

Table 3.5 lists all the CodeWarrior libraries you might need to add to 68K Pascal projects.

Table 3.5 Choosing libraries for a 68K Pascal application project

If the project…	Add this…
Is not for an MPW Tool or code resource	`PASCAL (`*`xx`*`).68K.lib`
Is for an MPW Tool	`PASCAL (`*`xx`*`).Tool.68K.lib`

If the project…	Add this…
Is for a code resource	`PASCAL (xx).A4.68K.lib`
Calls any math routines	`MathLib68K (2ixx).Lib`
Calls ANSI C routines	`MSL C xx.Lib` libraries, such as `MSL C.68K (4i).Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Uses console I/O	`MSL SIOUX.68K.Lib`

NOTE: In CodeWarrior Pascal, application projects must include Mathlib and an ANSI C library.

If the library's name in the table contains *xx*, you must choose a library based on some settings panel options. To learn how to choose a library by name, refer to "Library Naming Conventions" on page 421 to read about the naming conventions for Metrowerks libraries.

Power Macintosh Pascal Libraries

Table 3.6 lists all the CodeWarrior libraries you might need to add to PowerPC Pascal projects.

Table 3.6 Choosing libraries for a Power Macintosh application project

If the project…	Add this…
Is not for an MPW Tool	`MSL RuntimePPC.lib` and `PASCAL.PPC.lib`
Is for an MPW Tool	`MSL RuntimePPC.lib` and `PASCAL.Tool.PPC.lib`
Calls any math routines	`Mathlib`
Uses console I/O	`MSL SIOUX.PPC.Lib`

NOTE: In CodeWarrior Pascal, application projects must include Mathlib and an ANSI C library.

Setting Target Options

This section describes how to set the options in the application target settings panels, shown here in Figure 3.26 and Figure 3.27.

The topics covered in this section include:

- Setting the application's name
- Setting the application's type and creator
- Changing the memory requirements
- Setting the stack size
- Setting an application's 'SIZE' resource
- Changing the startup code

Figure 3.26 68K Application Project options

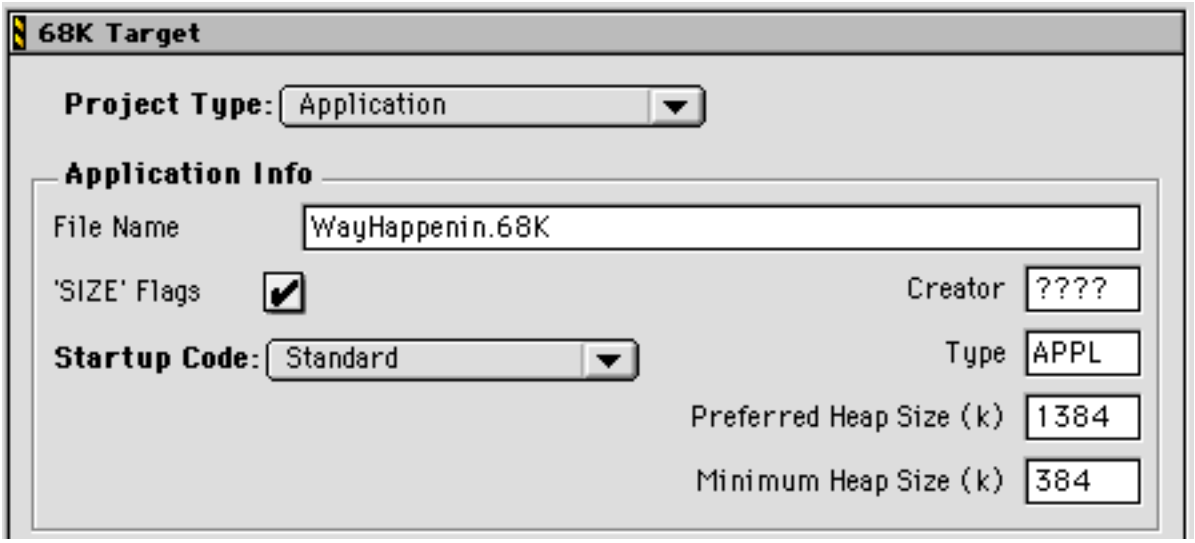
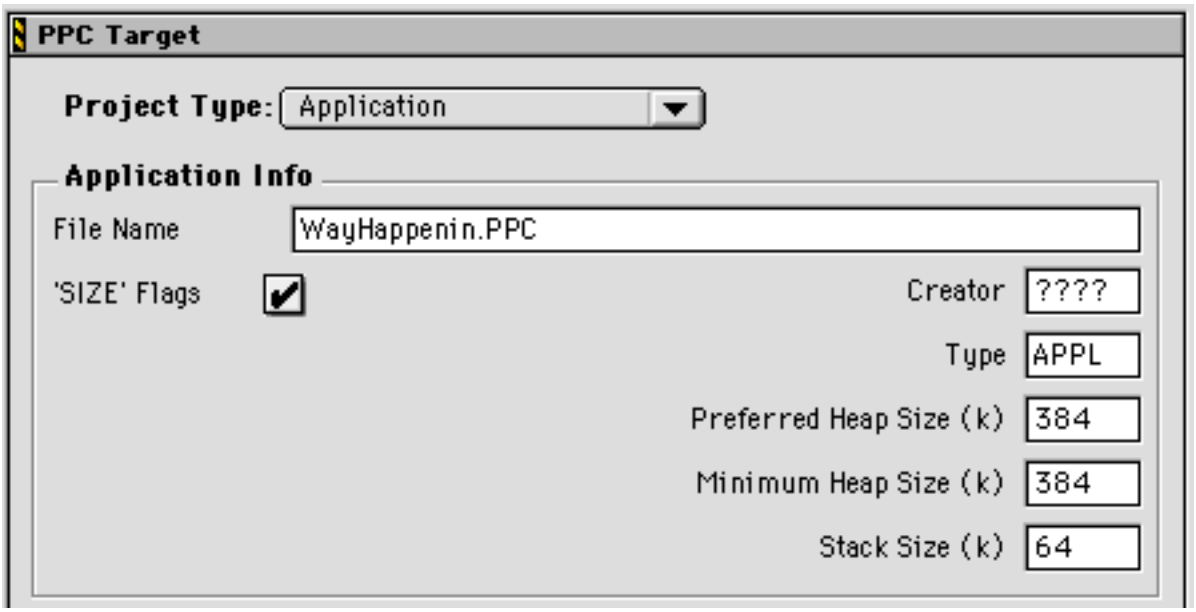


Figure 3.27 Power Macintosh and CFM68K Application Project options



Setting the application's name

When CodeWarrior creates your application, it gives the application file that is created at link time the name specified in the **File Name** field. You can change this field to give the file any name you want.

Setting the application's type and creator

The **Creator** and **Type** fields let you change the file type and creator of the linked output file that your project produces.

Understanding the Creator Code

The Finder identifies your application with its Creator code, a unique four-character sequence. Your application's Creator code must not conflict with any other application's.

NOTE: To be absolutely sure your Creator code is unique, register it with Apple Computer, Inc., at Macintosh Developer Technical Support. You should be able to find out more information about

this topic on the World-Wide Web at `http://devworld.apple.com`

Understanding the Type Code

An application's Type code must always be 'APPL', or 'appe'. The default value for this field is 'APPL'. If you use 'appe', you are writing a background application. Since there are some restrictions for what background applications can do, you should refer to other sources of information for applications of type 'appe'. One such source would be Apple Technical Notes, accessible on the World-Wide Web at `http://devworld.apple.com`

Other types of projects have different Type codes. For example, a code resource might have the Type 'INIT' or 'PRER', and a shared library might have the Type 'shlb'.

Changing the memory requirements

You can set how much memory your application has available to it in the heap when the user launches it, as shown in Table 3.7.

Table 3.7 Application memory sizes

This size…	Is…
`Preferred Heap Size`	The recommended amount of RAM your application wants. In the Finder's Get Info dialog, this is the number in the Suggested Size field and the default value for the Preferred Size field.
`Minimum Heap Size`	The minimum amount of RAM your application needs. In the Finder's Get Info dialog, this is the default value for the Minimum Size field.

Setting the stack size

An application's stack contains local variables, return addresses, and temporary information. If your application uses a lot of stack space (for example, it makes extensive use of recursive functions, or uses large arrays local to a routine), you may need to change the default size of the stack.

In a PowerPC or CFM68K Mac OS application, you can set the size in the Stack Size field. The default size is 64K.

In a 68K Mac OS application, you must set the stack size within your application's code by calling the function `SetApplLimit()`. For more information, see Chapter 1, "Introduction to Memory Management," in *Inside Macintosh: Memory*.

Setting an application's 'SIZE' resource

The flags in the 'SIZE' resource let the system software know which operating system features your application supports and expects. To set the flags, use the pop-up menu in Figure 3.28. Checked items are set to TRUE, unchecked items are set to FALSE. The menu may contain any number of checked items.

Figure 3.28 SIZE flags for application projects

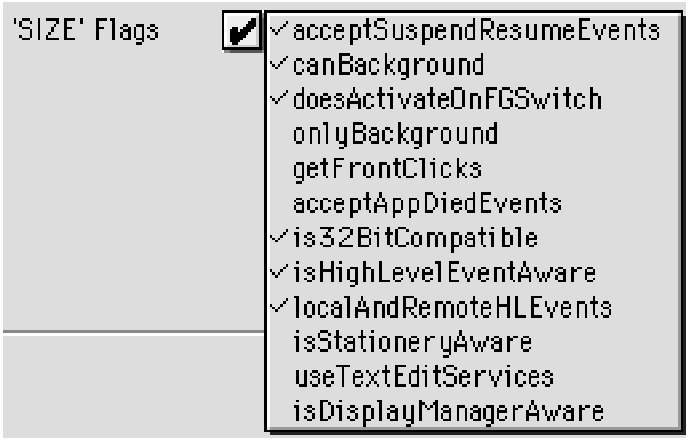


Table 3.8 describes the flags. For more information on these flags, with the exception of "isDisplayManagerAware," see the "Event

Manager" chapter in *Inside Macintosh: Macintosh Toolbox Essentials*. For more information on isDisplayManagerAware, see the "Display Manager" chapter in *Inside Macintosh: Advanced Color Imaging*.

Table 3.8 The 'SIZE' resource options

Turn this option on…	If your application…
`acceptSuspend-ResumeEvents`	Handles suspend and resume events as well as deactivate and activate events. These events are sent when your application is brought to the foreground or sent to the background.
`canBackground`	Supports null events when it is in the background. If you turn this option off, the Finder sends your application only update events.
`doesActiveOnFGSwitch`	Does not need to receive an activate (or deactivate) event after it has come to the foreground (or background).
`onlyBackground`	Contains no user interface and is meant to run in the background.
`getFrontClicks`	Supports front clicks which bring it to the foreground. Front clicks are mouse-down and mouse-up events.
`acceptAppDiedEvents`	Needs to know whether the application which it launched quit or crashed.
`is32BitCompatible`	Is 32-bit clean.

Turn this option on…	If your application…
`isHighLevelEventAware`	Supports Apple Events or other high level events. If you turn this option off, the Finder does not send high level events to your application.
`localAndRemoteHLEvents`	Handles Apple Events and other high-level events from remote computers. If you turn this option off, the Finder does not send your application AppleEvents from remote computers.
`isStationeryAware`	Supports stationery documents. If you turn this option off, the Finder handles stationery documents for you.
`useTextEditServices`	Supports inline text services introduced with System 7.
`isDisplayManagerAware`	Handles the Display Notice event, which tells your application to move its windows after the monitor settings have changed. If you turn this option off, the Display Manager moves your application's windows so they don't disappear off the screen.

Changing the startup code

By default, CodeWarrior adds code to your program that runs at startup to initialize static variables and handle other housekeeping tasks. The **Startup Code** pop-up menu lets you choose the startup code CodeWarrior adds at link time. The **Startup Code** options are shown in Table 3.9.

Table 3.9 The Startup Code options

This option…	Adds…
Standard	Standard start-up code.
The Debugger-Aware	Startup code that lets you use Steve Jasik's The Debugger to debug multiple-segment projects.
Custom	No startup code. You must create your own.

4

Creating Static Library Projects for Mac OS

A static library, sometimes just called a library, contains functions that applications, code resources, and other libraries can use. It does not execute on the computer's processor by itself. To use a library, the library must be included somehow in the build process of the program that is using it. The library must be linked with the application so that it becomes part of the code within the application.

NOTE: A shared or "dynamic" library is like a static library except that you don't have to link it into your main program's code to use it. Once you put a copy of a shared library on your system, any number of programs can use it. Refer to "Creating Shared Library Projects for Mac OS" on page 87 for more information.

You do not need to add MSL (Metrowerks Standard Libraries) source code or Mac OS Toolbox libraries to your library project. As long as you declare the Toolbox or MSL routines that you use as external using the `extern` keyword, CodeWarrior will link your library without any error. However, when you add this library to a project that creates an application or code resource, you must also add the required Toolbox libraries or MSL library or you will get link errors. In order to learn about which libraries to include in your project, refer to the chapter in this manual that discusses the type of code you are trying to build.

If you want to learn more about topics such as the `extern` keyword, there are a number of books available that cover the C language, such as *Learn C 2nd Edition* by Dave Mark.

This section describes:

- Types of Libraries
- Creating a Library Project
- Changing Library Project Options

Types of Libraries

CodeWarrior lets you create four types of libraries.

The different types of libraries are:

- 68K A5-based Application Libraries
- 68K A4-based Libraries
- CFM68K Libraries
- PowerPC Libraries

To learn more about the 68K processor A5 register and its memory management role in the Macintosh 68K runtime model, refer to the book *Inside Macintosh: Memory*, published by Addison-Wesley.

68K A5-based Application Libraries

A 68K A5-based application library is used primarily for linking with 68K applications. It references global data from the A5 register of the 68K processor. The A5 register is typically used to reference global variables from, with a construct typically called an A5 World.

68K A4-based Libraries

A 68K A4-based library is used primarily for 68K code resources. It references global data from the A4 register, instead of the A5 register. It is typically used for programs that should not disturb an A5 run-time environment. This typically arises when using standalone code, like code resources, because there is no A5 World set up for standalone code modules.

CFM68K Libraries

A CFM68K library is used for CFM68K applications. It is not possible to create CFM68K code resources.

To learn more about the Code Fragment Manager, refer to the book *Inside Macintosh: PowerPC System Software*, published by Addison-Wesley.

PowerPC Libraries

A PowerPC library can be used for any PowerPC Mac OS program.

Creating a Library Project

This section describes how to create a 68K or PowerPC library project.

If you would like to learn about how to set up multiple targets for your project, you should refer to the IDE User Guide documentation on the CodeWarrior CD. Complex Projects that contain multiple targets are often useful for generating debug and shipping versions of your code from the same project.

The topics discussed in this section include:

- Creating a 68K Mac OS Library
- Creating a PowerPC Mac OS Library

Creating a 68K Mac OS Library

Here are the steps you need to follow to create a Mac OS 68K library project.

1. **Create an empty project.**

   To do this, choose the **New Project** command in the **File** menu, and select Empty Project from the stationery dialog box that is presented, as shown in Figure 4.1. Then give your library project a name and save it in the desired location on your hard disk, using

the dialog box as shown in Figure 4.2. You should see your new project window appear on the screen.

Figure 4.1 Creating an Empty Project

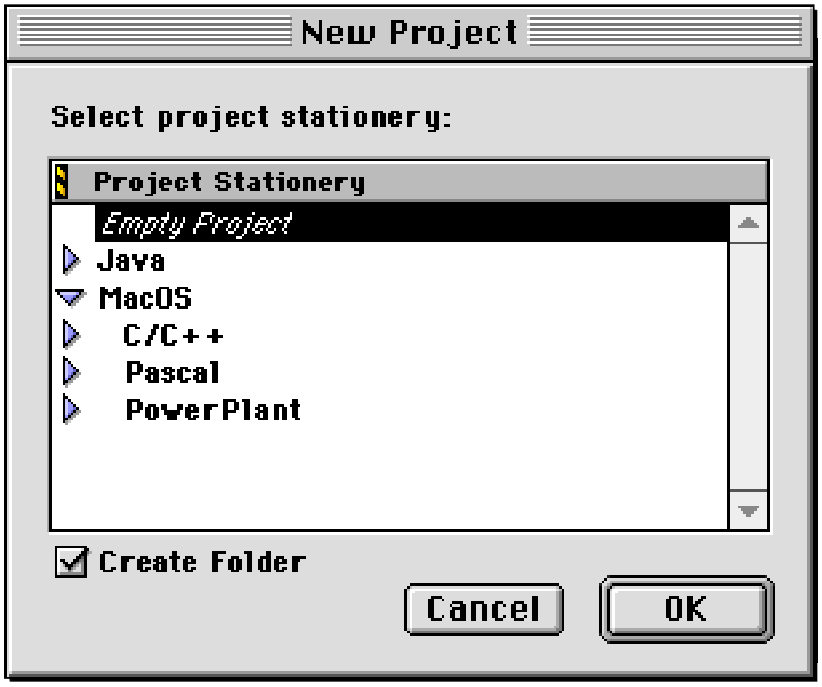
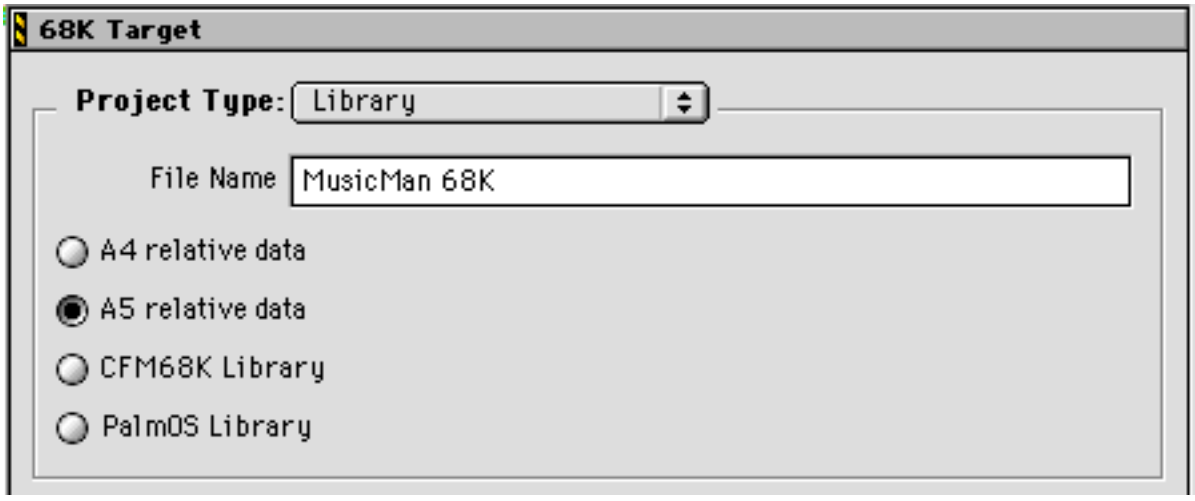


Figure 4.2 Project Naming Dialog Box

2. **Set the Project type to a 68K library.**

   Choose **Preferences** from the **Edit** menu. Click on the **68K Target** panel in the list on the left side of the dialog box that is shown, then select **Library** from the **Project Type** pop-up menu, as shown in Figure 4.3.

Figure 4.3 Selecting Library Project Type for 68K



3. **Choose the library type.**

   You should choose either A4 Relative Data or A5 Relative Data from the radio buttons in the dialog box, depending on whether you want to have your data referenced from the 68K A4 or A5 register.

   If the library is for use with applications, turn on the A5-Relative Data option. The library will reference its global data from the A5 register.

   If the library is for use with code resources, turn on the A4-Relative Data option. The library will reference its global data from the A4 register.

   To learn more about where to go for more information on A4 and A5 register selection, refer to "Types of Libraries" on page 76.

   The CFM68K Library radio button is explained in the section titled "Creating Shared Library Projects for Mac OS" on page 87.

4. **Enter the name for the library in the File Name field.**

   When CodeWarrior creates your library, it gives it the default name specified in the **File Name** field. You should change this to the name you want your library file to have.

5. **Add files to your project, choose any options settings you want, and build it.**

   To learn how to add files to your project, change settings such as code generation, and build it, refer to the *IDE User Guide* on your CodeWarrior CD. To learn detailed information on how to change the settings for parameters like code generation and debugging for your library, refer to "Target Settings for 68K" on page 249.

Creating a PowerPC Mac OS Library

Here are the steps you need to follow to create a Mac OS PowerPC library project.

1. **Create an empty project.**

   To do this, use the **New Project** command in the **File** menu, and select **Empty Project** from the stationery dialog box that is presented,

as shown in Figure 4.1 on page 78. Then give your library project a name and save it in the desired location on your hard disk, using the dialog box as shown in Figure 4.2 on page 79. You should see your new project window appear on the screen.

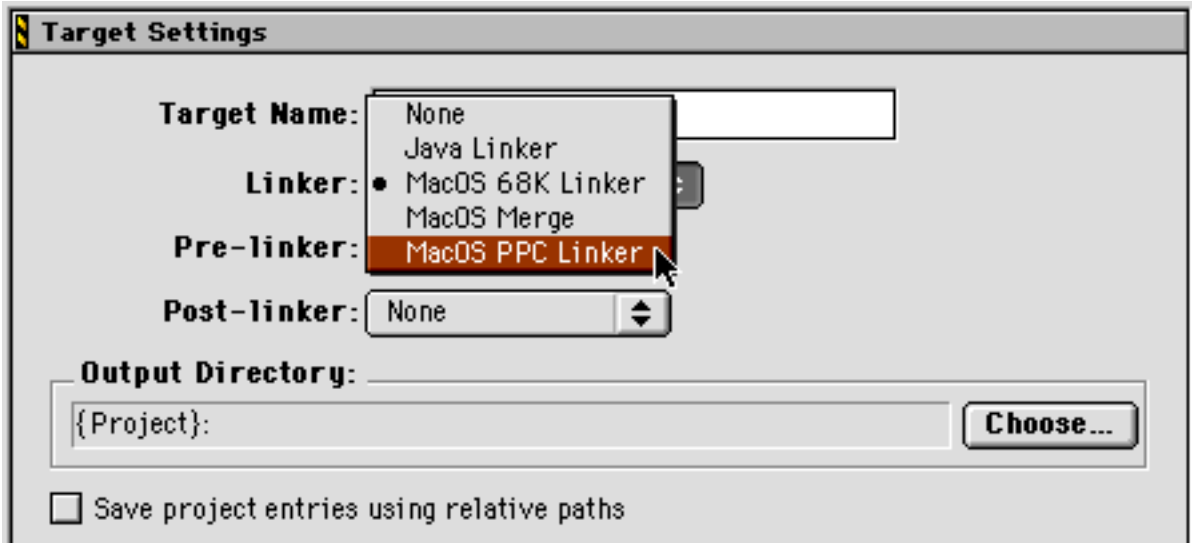
2. **Bring up the Target Settings panel.**

   Choose ***MyTarget*** **Settings** from the **Edit** menu, where ***MyTarget*** is the name of the current active target in the frontmost project.

   Click **Target Settings** from the list on the left side of the dialog box.

3. **Set the Target linker to the PowerPC linker.**

Figure 4.4 Selecting the PowerPC Linker



Choose **Mac OS PPC Linker** from the **Linker** pop-up menu in the dialog box, as shown in Figure 4.4. When you do this, a dialog box will appear as shown in Figure 4.5. Click on **OK** to dismiss this dialog box.

Figure 4.5 Must Be Rebuilt

Target "CoolApp" must be rebuilt.
Changes to the settings of target "CoolApp" require that its sources must be recompiled. Do you want to continue?
Cancel
OK

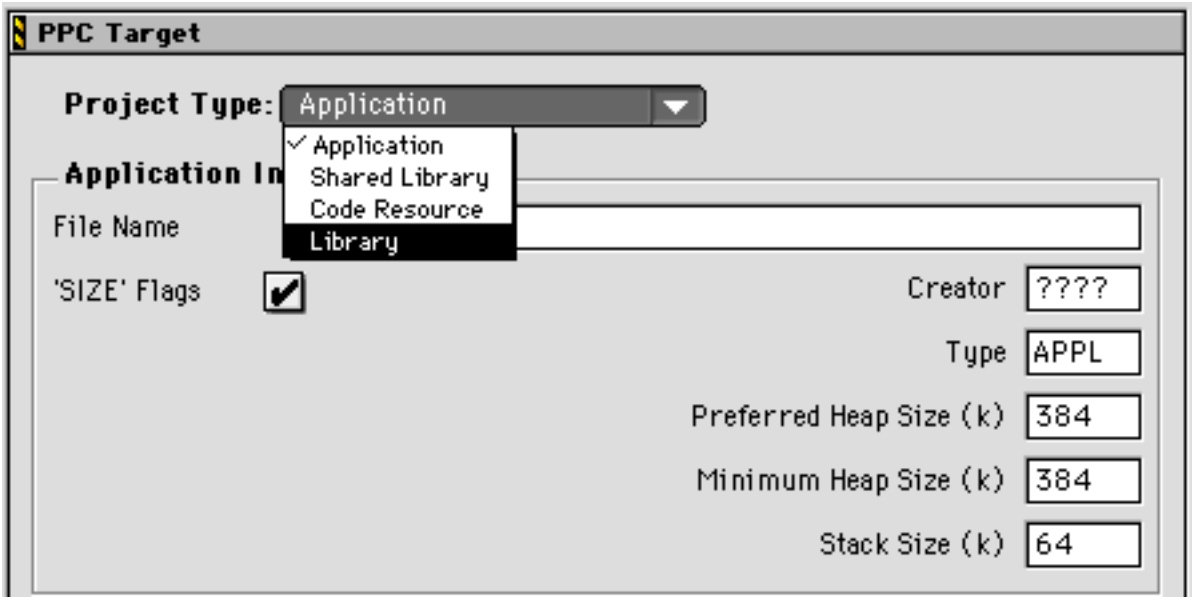
4. **Choose the PPC Target settings.**

   Click on the PPC Target text in the left side of the dialog box. The user interface on the right side of the dialog box will change, so that you can do the next step.

5. **Choose the Library Project Type.**

   Choose **Library** from the **Project Type** pop-up menu, as shown in Figure 4.6. The user interface will change to show you the relevant options for the next step.

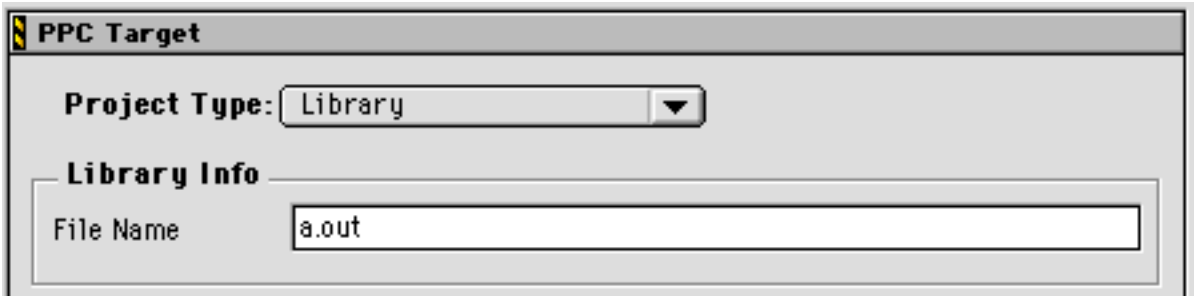
Figure 4.6 Choosing the Library Project Type



6. **Enter the name for the library in the File Name field.**

   When CodeWarrior creates your library, it gives it the default name specified in the **File Name** field, as shown in Figure 4.7. You should change this to the name you want your library file to have.

Figure 4.7 Setting the File Name for your library



7. **Add files to your project, choose any options settings you want, and build it.**

   To learn how to add files to your project, change settings such as code generation, and build it, refer to the *IDE User Guide* on your CodeWarrior CD. To learn detailed information on how to change the settings for parameters like code generation and debugging for your library, refer to "Target Settings for PowerPC" on page 169.

Changing Library Project Options

This section describes how to change the options in the library project preference panel after you have already created a project. If you haven't created a project yet, refer to "Creating a Library Project" on page 77 to learn how.

To learn detailed information on how to change the settings for parameters like code generation and debugging for your library, refer to "Target Settings for PowerPC" on page 169 or "Target Settings for 68K" on page 249.

68K Libraries

If you want to change from using A4 to A5-relative addressing, you can use the **68K Target** panel, as shown in Figure 4.8 to do this. To bring up this panel, choose **Target Settings** from the **Edit** menu, then select **68K Target** from the list on the left side of the dialog box that appears.

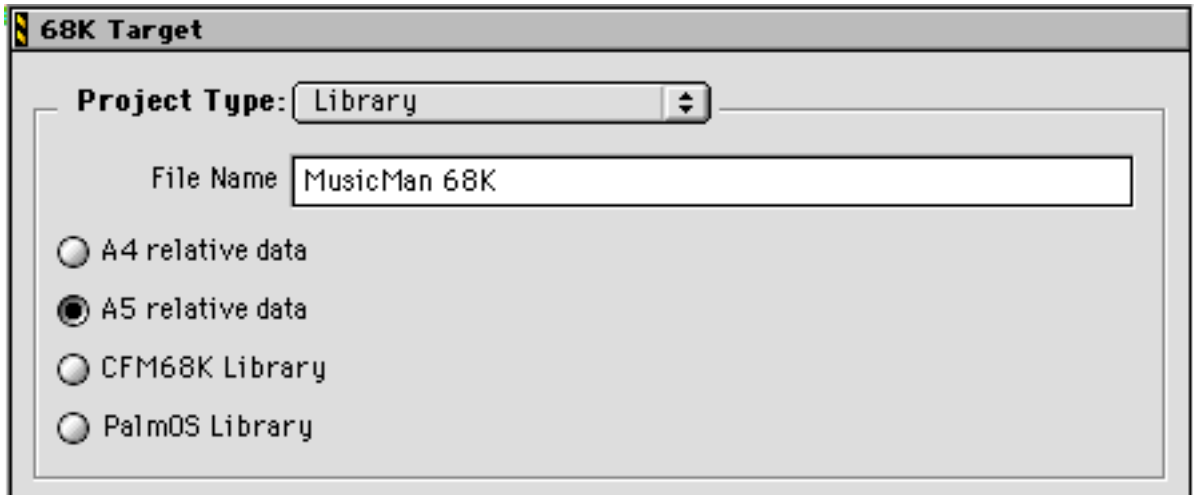
You can switch to A5 or A4 addressing by clicking the radio button that corresponds to your choice. Note that if you have written inline assembly code, using the `asm` keyword, that references data from A4 or A5, you will need to modify your code.

You can also change the name of your library if desired.

After making your choice, make sure to rebuild your project.

The CFM68K Library radio button is explained in the section titled "Creating Shared Library Projects for Mac OS" on page 87.

Figure 4.8 68K Library Project options



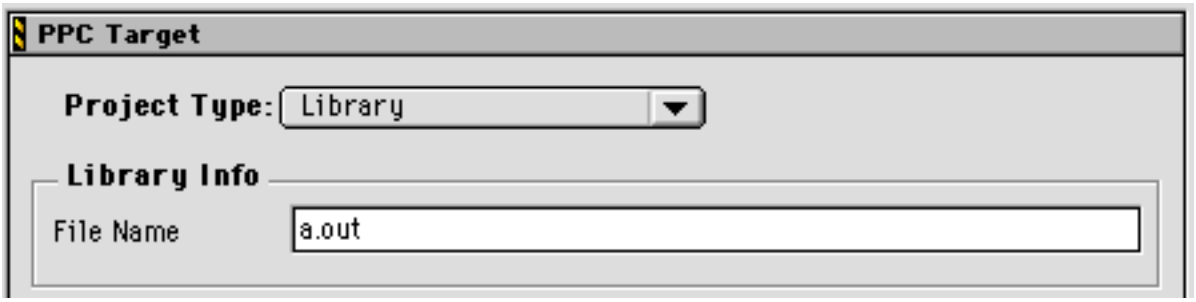
PowerPC Libraries

If you want to change the name of your PowerPC library, first open the **PPC Target** panel, as shown in Figure 4.9. To open this panel,

choose **Target Settings** from the **Edit** menu, then select **PPC Target** from the list on the left side of the dialog box that appears.

You can then change the name of your library if desired. After making your choice, make sure to rebuild your project.

Figure 4.9 PowerPC Library Project options



5

Creating Shared Library Projects for Mac OS

A shared library contains routines that applications, code resources, and other libraries can use. It does not run by itself. Unlike a static library, it isn't necessary to compile the shared library directly in your programs. Shared libraries are dynamically loaded into applications at runtime by the Mac OS.

This section describes:

- Important Information on Shared Libraries
- Types of Shared Libraries
- Creating a Shared Library
- Setting Shared Library Project Options
- Choosing Libraries for a Shared Library Project

See also Apple's documentation on the Mac OS Runtime architecture found at:

```
http://gemma.apple.com/deve/techsupport/insidemac/
runtimehtml/RTArch-2.html
```

Important Information on Shared Libraries

There are a few things to you need to know when creating a shared library. In particular, you need to know if the shared library requires its own copy of the C++ runtime data structures, especially the exception handling code, or whether it will share it with other fragments in the same application. In a multi-fragment application, you want only one copy of the C++ runtime data structures and

support routines, but a stand-alone shared library will need to have its own copy.

If you want to write a shared library that can be used by any application, you need to make sure all exceptions thrown in the shared library are caught by the shared library. You should also arrange for the library's own static initialization and termination. This kind of shared library is called a *monolithic* or *stand-alone* shared library. You must link with a runtime library that will give the shared library its own copy of all the C++ runtime data structures.

If you are writing a large application, you may want to break it up into smaller applications and one or more shared libraries for organizational purposes. This is a *multi-fragment* application. In this case, you would need to have the C++ runtime code in its own shared library that is linked to the other shared libraries. The application project should link against all shared libraries. The code from each shared library will behave the same as if it had all been statically linked into your application. Exceptions can be thrown freely from one shared library to anther and the destructors for static variables in the shared libraries get called at the proper times (as the application is exiting and before `stdio` is cleaned up).

For information on how to choose libraries for a shared library project, refer to "Choosing Libraries for a Shared Library Project" on page 96.

Types of Shared Libraries

You can create two types of shared libraries with CodeWarrior:

- CFM68K shared libraries
- PowerPC shared libraries

CFM68K shared libraries

CFM68K shared libraries are only used with CFM68K applications. You can not use these with classic 68K or PowerPC programs. Applications using a CFM68K shared library must be built as a CFM68K application, will only run on a 68K Macintosh (not a Pow-

erPC-based Macintosh), and require the `CFM-68K Runtime Enabler` extension to be installed.

PowerPC shared libraries

PowerPC shared libraries are used with a PowerPC Macintosh. You cannot use these with 68K or CFM68K applications.

Creating a Shared Library

This section describes how to create a 68K or PowerPC Mac OS shared library project using C, C++ and PowerPlant, or Pascal. The only difference between creating a C/C++ shared library and a Pascal shared library are the choice of library files you need to add in your project.

NOTE: Shared libraries are sometimes also known as Import Libraries or Dynamically Linked Libraries (DLL's), and sometimes end in "DLL".

The topics discussed in this section are:

- Creating a CFM68K Shared Library
- Creating a PowerPC Shared Library
- Creating a Fat Shared Library
- Exporting Symbols

Creating a CFM68K Shared Library

Follow these steps to create a shared library for use with CFM68K applications. Note that any 68K computer you want to use the shared library on with need to have the software extension called `CFM-68K Runtime Enabler` installed. This is required for Mac OS 7.6.1 and older.

1. **Create an empty project.**

   Choose the **New Project** option from the **File** menu, and select the **Empty Project** option in the dialog box that appears.

2. **Configure the project settings.**

   In the **Target Settings** panel, choose Mac OS 68K Linker from the Linker pop-up menu.

   In the **68K Target** panel, choose **CFM68K Shared Library** from the **Project Type** menu.

   For more information on configuring shared library project options, refer to "Setting Shared Library Project Options" on page 95.

3. **Set project options.**

   For more information on the shared library project options, see "Setting Shared Library Project Options" on page 95.

4. **Choose the symbols to export, if any.**

   For more information on how to export symbols, refer to "Exporting Symbols" on page 92.

5. **Add your source code files to the project.**

   Use drag and drop or the **Add Files** menu command from the **Project** menu to add source code files to your project.

6. **Add library files to the project.**

   To learn about which library files to add to your project, refer to "Choosing Libraries for a Shared Library Project" on page 96.

Creating a PowerPC Shared Library

This procedure allows you to create a shared library for use with PowerPC applications.

1. **Create an empty project.**

   Choose the **New Project** command on the **File** menu, and select the **Empty Project** option in the dialog box that appears.

2. **Configure the project settings.**

   In the **Target Settings** panel, choose **Mac OS PPC Linker** from the **Linker** pop-up menu.

   In the **PPC Target** panel, choose **Shared Library** from the **Project Type** menu.

   For more information on configuring shared library project options, refer to "Setting Shared Library Project Options" on page 95.

3. **Set project options.**

   For more information on the shared library project options, see "Setting Shared Library Project Options" on page 95.

4. **Choose the symbols to export, if any.**

   For more information on how to export symbols, refer to "Exporting Symbols" on page 92.

5. **Add your source code files to the project.**

   Use drag and drop or the **Add Files** menu command from the **Project** menu to add source code files to your project.

6. **Add library files to the project.**

   To learn about which library files to add to your project, refer to "Choosing Libraries for a Shared Library Project" on page 96.

Creating a Fat Shared Library

Follow these steps to create a Fat shared library for use with CFM68K and PowerPC applications. Note that any 68K computer you want to use the shared library on need to have the software extension called `CFM-68K Runtime Enabler` installed. This is required for Mac OS 7.6.1 and older.

1. **Create two projects, one for CFM68K and one for PowerPC.**

   To learn how to create these two projects, refer to "Creating a CFM68K Shared Library" on page 89, and "Creating a PowerPC Shared Library" on page 90.

2. **Add the CFM68K output binary to the PowerPC project.**

   To do this, you can use drag and drop to drag the CFM68K library and drop it onto the Files view of the PowerPC project window.

3. **Set the Merge Into Output options.**

   Click on the CFM68K binary file in the PowerPC project window to select it, then bring up the Project Inspector window (on the Window Menu) and click on Merge into Output to turn this option on.

4. **Rebuild your PowerPC project.**

   Rebuild your project and verify that your PowerPC library output file now contains 68K code. To verify, use a resource editor like ResEdit or Resorcerer to confirm the presence of CFM68K and PPC member in the `'cfrg'` resources in the resource fork of the library.

Exporting Symbols

When you create a shared library, you must specify which symbols to export; that is, specify which symbols can be used by other applications and libraries that will need to link against your shared library.

CodeWarrior gives you several ways to export symbols:

- Exporting symbols with an .exp file
- Exporting symbols from a shared library with a #pragma option
- Exporting all globals in a shared library
- Do not export globals in a shared library

Exporting symbols with an .exp file

This method allows you to export symbols to a file that has a `.exp` extension. This .exp file has the same name as your project. For example, if your project named is `MyShLib.mcp`, the `.exp` file would be named `MyShLib.mcp.exp`.

Using the .exp file option is a three step process. First you create the `.exp` file. You then edit the file to remove any symbols you don't

want. Third, you add the `.exp` file back into your project and compile again.

1. **Select the proper project settings.**

   In the **PPC PEF** or **CFM68K** panel, select **Use .exp** file from the **Export Symbols** pop-up menu.

2. **Delete any .exp file that is already in your project's folder.**

   CodeWarrior will not create a new export file if one already exists in your project's folder.

3. **Make your project.**

   Choose **Make** from the **Project** menu.

   CodeWarrior creates an export file, which lists the names of all the global variables and routines in every one of your project's files. This file's name is your project's file name with `.exp` appended.

4. **Examine the export file with the editor.**

   Choose the **Open** command from the **File** menu to open the `.exp` file. Listing 5.1 shows a sample .exp file.

Listing 5.1 Sample export file

```
FindDominantDevice__8UWindowsFRC4Rect#
UWindows::FindDominantDevice(const Rect &)
GetWindowStructureRect__8UWindowsFP8GrafPort#
UWindows::GetWindowStructureRect(GrafPort *)
GetWindowContentRect__8UWindowsFP8GrafPort#
UWindows::GetWindowContentRect(GrafPort *)
# __sinit
# __local_destructor_chain
# __global_destructor_chain
# __dc_arr
# __del_arr
# __new_arr
# __init_arr
# __copy
# __destroy_global_chain
# __destroy_local_chain
# __destroy_local_objects_to
```

```
# __destroy_local_objects
# __reregister_local_object
# __preregister_local_object
# __register_local_object
# __register_global_object
# __ptmf_cast
```

5. **Comment out any symbol you don't want to export.**

   To comment out a symbol, put a # at the beginning of the line that contains it.

6. **Comment out any CodeWarrior runtime symbol.**

   CodeWarrior runtime symbols begin with two underscores (__).

   If you do not comment these symbols, CodeWarrior may generate linker errors when you use this library in other projects.

7. **Add the .exp file and recompile**

   Add the `.exp` file back into your project. Assign it to the proper target if you have more than one and recompile.

Exporting symbols from a shared library with a #pragma option

This method allows you to export symbols using a `#pragma`.

1. **Select the proper project settings.**

   In the **PPC PEF** or **CFM68K** settings panel, select the **Use `#pragma`** option from the **Export Symbols** pop-up.

2. **In your source files, turn on the export pragma for all the routines you want to export.**

   When the export pragma is on, CodeWarrior exports any function your declare or define. For example, this code exports `DrawBear()`, `DrawCat()`, `DrawLizard()`, and `DrawAlligator()`, but `DrawMammal()` and `DrawReptile()` are internal.

```
#pragma export on
OSErr DrawBear(void);
OSErr DrawCat(void);
OSErr DrawLizard(void);
```

```
OSErr DrawAlligator(void);
#pragma export off

OSErr DrawMammal(void);
OSErr DrawReptile(void);
```

Exporting all globals in a shared library

This method allows you to export all symbols in a shared library.

In the **PPC PEF** or **CFM68K** panel, choose **All Globals** from the **Export Symbols** pop-up.

When you build you project, all the global symbols in your project are exported.

Do not export globals in a shared library

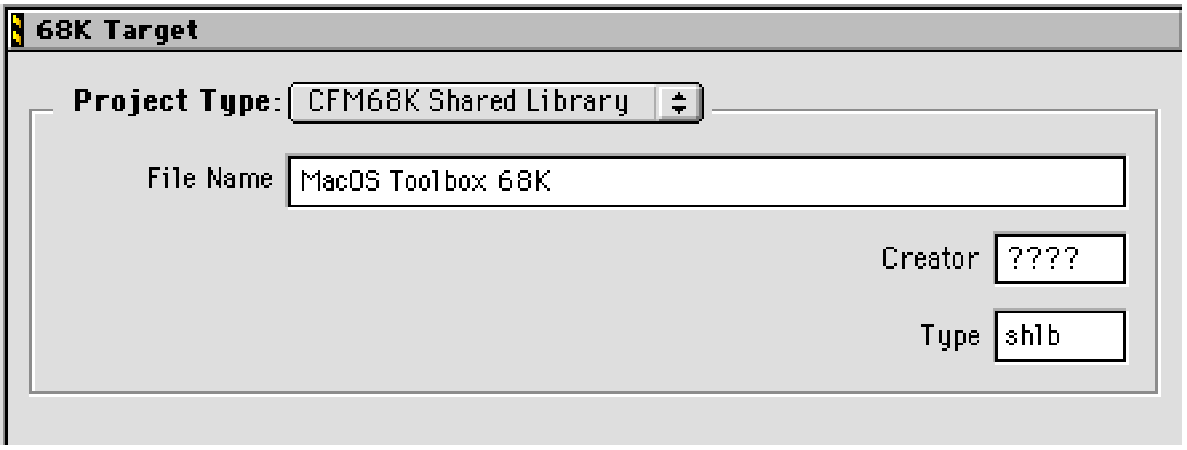
In the **PPC PEF** or **CFM68K** panel, choose **None** from the **Export Symbols** pop-up.

When you build your project, none of the global symbols in your project are exported.

Setting Shared Library Project Options

This section describes how to configure the options in the shared library Target Settings panel, shown in Figure 5.1. You can access this panel by choosing **PPC Target** or **68K Target** from the list of panels on the **Target Settings** dialog box, which you open by choosing ***Target* Settings** from the **Edit** menu in the IDE.

Figure 5.1 Power Macintosh and CFM68K Shared Library Project options



Setting the shared library's name

When CodeWarrior creates your shared library at link time, it gives it the name specified in the File Name field. You can change this name to suit your purposes.

Setting the library's type and creator

The **Creator** and **Type** fields specify the creator and type codes for the shared library. The proper values for these fields depend on what kind of library you're creating.

The **Type** field lets an application or the system software identify what the file contains. By default, the CodeWarrior IDE sets this field to `'shlb'`, which is for a shared library.

Choosing Libraries for a Shared Library Project

Table 5.1 details the libraries you need to include in your project in order to create a shared library.

Table 5.1 Special files for shared libraries

If the library…	For PPC add...	For CFM68K add...
Calls any Macintosh Toolbox routines	`InterfaceLib`	`InterfaceLib`
Uses Pascal	`PASCAL.PPC.Lib` (some routines of this library also need an MSL library, `MathLib`, and `InterfaceLib`)	`Not Supported`
For Pascal using objects, and deriving classes from TObject	`ObjIntf.p`	`Not Supported`

TIP: To simplify the use of Pascal objects within shared libraries, turn on Optimize Class Hierarchy in the **Pascal Language** panel, as no fragment initialization code is required to support class initializations.

In addition, depending on the kind of shared library you are writting, you may need to add the libraries shown in Table 5.2. For example, if you are using a Monolithic app, you will need to include MSL RuntimePPC.Lib.

NOTE: Monolithic means that the runtime libraries are statically linked into projects that create shared libraries or application and drop-ins. Multi-fragment shared libraries, applications and drop-ins are so called because they all share a single runtime library that is a shared library itself.

Table 5.2 Choosing special purposes runtime libraries

	MSL App-Runtime.Lib	MSL ShLib-Runtime.Lib	MSL DropIn-Runtime.Lib	MSL Runtime-PPC.Lib	MSL Runtime-PPC.DLL	MSL Runtime-PPC++.DLL
Monolithic app				X		
C Monolithic shlb or drop-in				X		
C++ Monolithic shlb or drop-in				X		
C Multi-fragment app	X				X	
C++ Multi-fragment app	X					X
C Multi-fragment shlb		X			X	
C++ Multi-fragment shlb		X				X
C Multi-fragmenet drop-in			X		X	
C++ Multi-fragment drop-in			X			X

For more information, refer to "Using Libraries" on page 391 and "Runtime Libraries" on page 397.

Difference Between Shared Libraries and Drop-Ins

A drop-in shared library is distinguished from a normal shared library in that it may be loaded and unloaded several times by the application. An example of a drop-in shared library would be a Photoshop™ plug-in.

A normal shared library (one that is linked with your program) is loaded implicitly by the Mac OS when your application starts up. A Drop-In shared library is found and loaded by your program explicitly.

Solving Common Problems

Often, when creating a shared library, you will encounter warning messages from the Mac OS Linker such as those in Listing 5.2. These warnings are a product of the CodeWarrior runtime architecture and can be safely ignored.

Listing 5.2 Common warning messages

```
Link Warning : ignored '__register_global_object' (descriptor) in
  MSL RuntimePPC++.DLL
Previously defined in MSL DropInRuntime.Lib

Link Warning : ignored '__destroy_global_chain' (descriptor) in
  MSL RuntimePPC++.DLL
Previously defined in MSL DropInRuntime.Lib

Link Warning : ignored '__global_destructor_chain' (data) in
  MSL RuntimePPC++.DLL
Previously defined in MSL DropInRuntime.Lib
```

6

Creating Code Resource Projects for Mac OS

Mac OS code resources are binary files that add functionality to an application or the system software but that cannot run by themselves. Many applications, such as Photoshop™, HyperCard™, and 4th Dimension™, let you add features and commands with special types of code resources. The Mac OS lets you add functionality with such code resources as Chooser extensions, Control Panels, Window Definitions (WDEFs), and Control Definitions (CDEFs), among others.

This section describes:

- Types of Code Resources
- Creating a Code Resource
- Creating 68K Single or Multi Segment Resources
- Setting Code Resource Options
- Choosing Libraries for Code Resource Projects
- Writing Code Resources

Types of Code Resources

You can create many kinds of code resources for both 68K and PowerPC-based Mac OS computers. There are many different types of code resources. This section will discuss the various types, and give you some overview information about them.

NOTE: CodeWarrior does not support the automatic creation of PowerPC device drivers.

The following types of code resources run on Mac OS computers:

- 68K Device Driver
- 68K Standard Code Resource
- 68K Custom Code Resource

The following types of PowerPC code resources can only run on a PowerPC-based Mac OS computer:

- PowerPC Private Code Resource
- PowerPC Accelerated Code Resource

The following code resource types contain both 68K and PowerPC code.

- Fat Code Resource
- Safe Fat Code Resource

68K Device Driver

A 68K Device Driver code resource is a code resource that is used for interfacing to hardware devices, such as Ethernet networking cards. When linking a device driver code resource, the CodeWarrior linker automatically adds a a null character to the beginning of your resource's name and adds a device driver header to your resource when you build it.

68K Standard Code Resource

A standard code resource is perhaps the most common type of code resource. One example of a standard code resource is a Photoshop plug-in. The CodeWarrior linker adds a standard header to your resource when you build it.

68K Custom Code Resource

You may also create a custom code resource. The CodeWarrior linker doesn't add a header to your custom resource. You must create one on your own.

PowerPC Private Code Resource

You may create a PowerPC private code resource that your application calls directly. The CodeWarrior linker adds no start-up code to this type of resource. To create one, choose None from the Header menu.

PowerPC Accelerated Code Resource

You can create a PowerPC accelerated code resource, that someone else's application or the system software calls. The CodeWarrior Linker adds start-up code to your resource that allows a 68K application, running under the emulator on a PowerPC, to use the resource. The resource runs only on a PowerPC Mac OS computer. To create one, choose Native from the Header menu.

Fat Code Resource

You can create a Fat code resource runs only on a PowerPC Mac OS computer. A Fat code resource contains a header that allows the Mac OS Mixed Mode Manager to avoid a context switch by letting the resource run either native or under the 68K emulator.

Safe Fat Code Resource

You can create a safe Fat code resource, which runs on any Mac OS computer. It contains a header that decides whether the resource is running on a 68K or a PowerPC, and calls the correct code for the Mac OS computer that it is running on.

Creating a Code Resource

This section describes how to create projects to build 68K and PowerPC code resources.

The topics in this section include:

- Creating a 68K Code Resource Project
- Creating a PowerPC Code Resource Project

Creating a 68K Code Resource Project

Here are the steps needed to create a 68K code resource project.

1. **Create an empty project.**

   To create a new empty project, choose **New Project** from the File Menu. The New Project dialog box is displayed, as shown in Figure 6.1. Choose Empty Project and click OK, then give it a name in the dialog box that appears, as shown in Figure 6.2.

Figure 6.1 New Project window

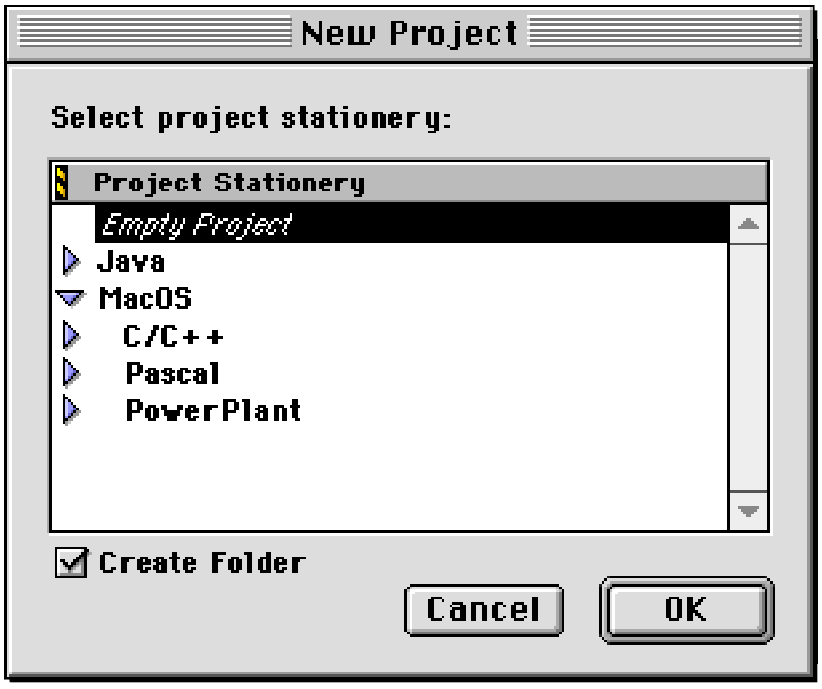
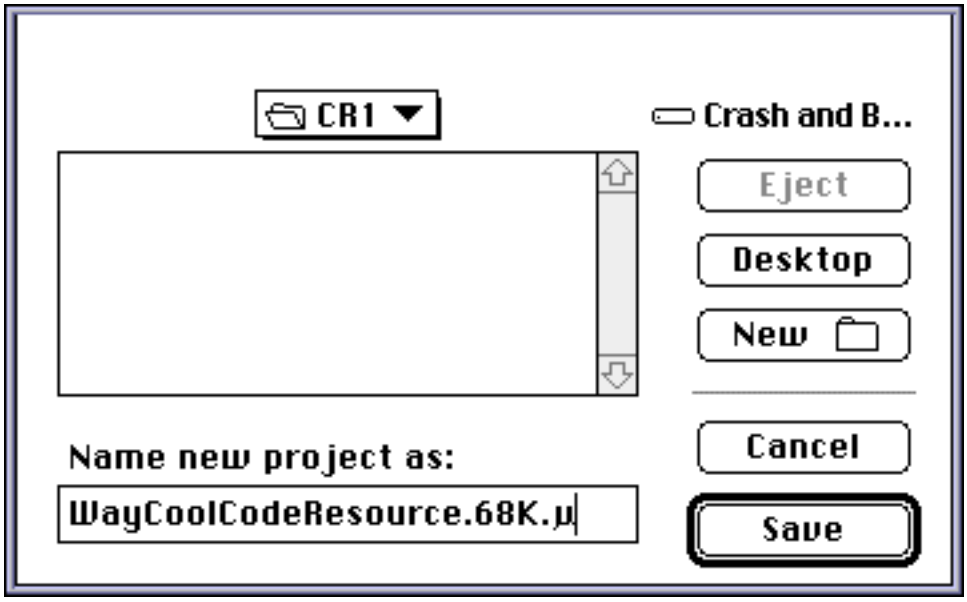


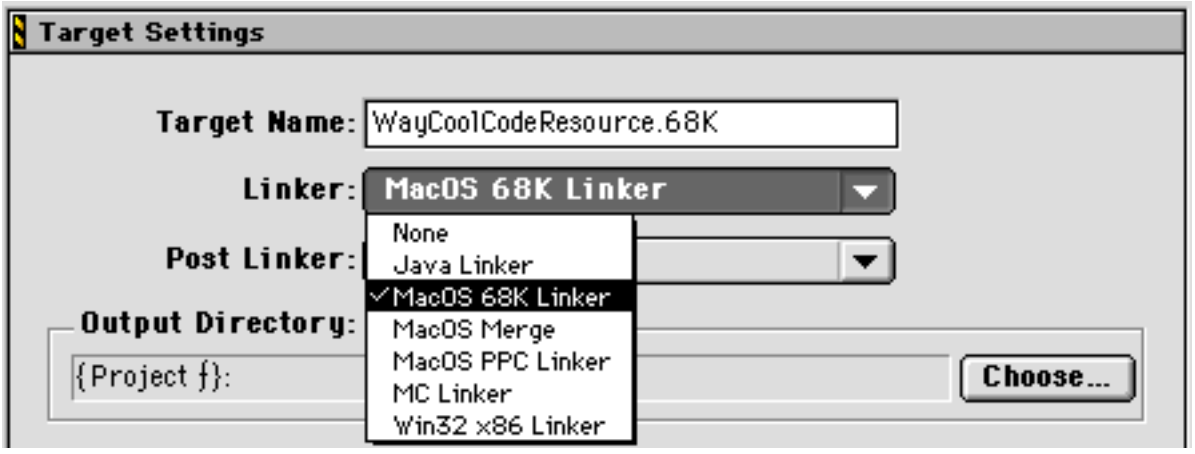
Figure 6.2 New Project Naming



2. **Select the 68K Linker.**

   This may already be set for you, but use the Target Settings command under the Edit Menu (the command will be renamed for your project) to show the **Target Settings** panel, as shown in Figure 6.3. Make sure that Mac OS 68K Linker is showing in the Linker pop-up menu.

Figure 6.3 68K Linker Choice in Target Settings



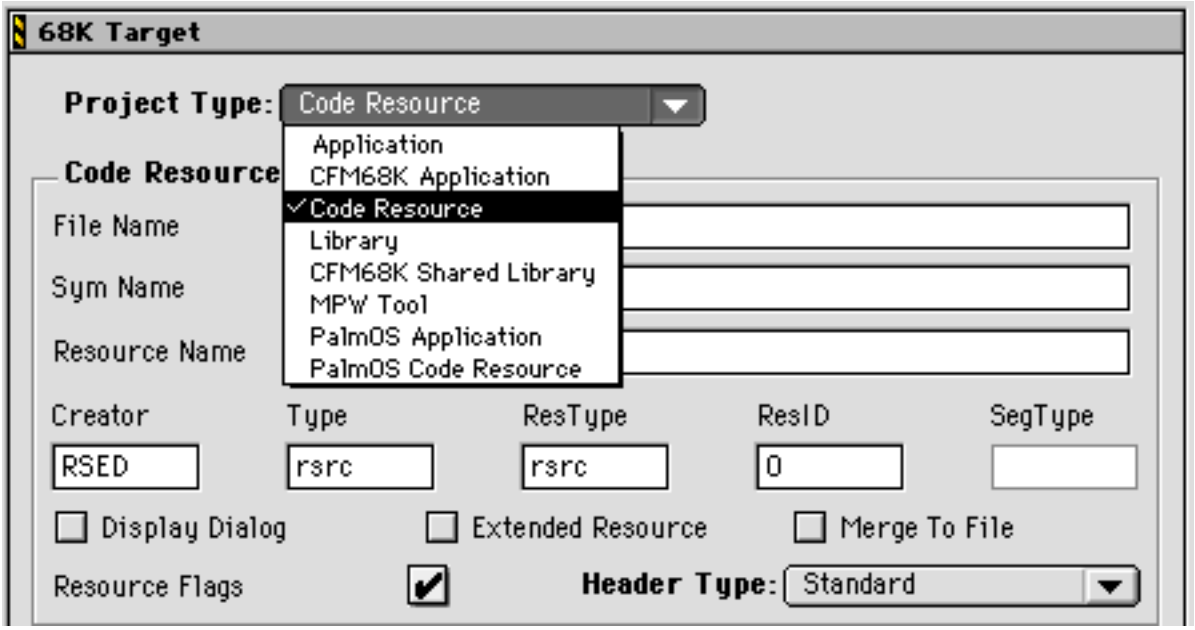
3. **Give your target a name.**

   Type a name into the Target Name field, as shown in Figure 6.3

4. **Choose Code Resource from the Project Type menu.**

   Use the **68K Target** to choose the Code Resource Project Type from the pop-up menu. This is shown in Figure 6.4.

Figure 6.4 Choose Code Resource Project Type



5. **Give your code resource a name using the File Name field in the panel shown in Figure 6.4.**

   You can also change the Creator, Type, ResType, ResID, SegType, Header Type, and other parameters at this point.

6. **Add all the needed libraries to your project.**

   For more information on selecting libraries to add to your project, see "Choosing Libraries for Code Resource Projects" on page 121.

7. **Choose whether it will be a single-segment or multi-segment resource.**

   To learn how to set this parameter, refer to "Creating 68K Single or Multi Segment Resources" on page 110.

8. **Set project options.**

   You can set the options for code generation and debugging at this time. To learn how to do this, refer to "Setting Code Resource Options" on page 114.

Creating a PowerPC Code Resource Project

Here are the steps needed to create a PowerPC code resource project.

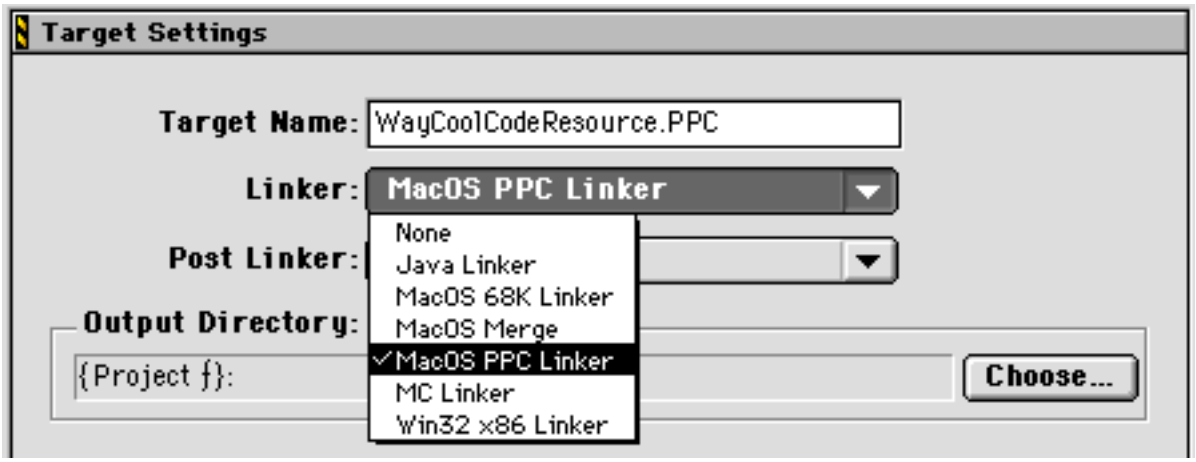
1. **Create an empty project.**

   To create a new empty project, choose New Project from the File Menu. The New Project dialog box is displayed, as shown in Figure 6.1 on page 104. Choose Empty Project and click OK, then give it a name in the dialog box that appears, as shown in Figure 6.2.

2. **Select the PPC Linker.**

   Use the Target Settings command under the Edit Menu (the command will be renamed for your project) to show the **Target Settings** panel, as shown in Figure 6.5. Make sure that Mac OS PPC Linker is showing in the Linker pop-up menu.

Figure 6.5 PPC Linker Choice in Target Settings



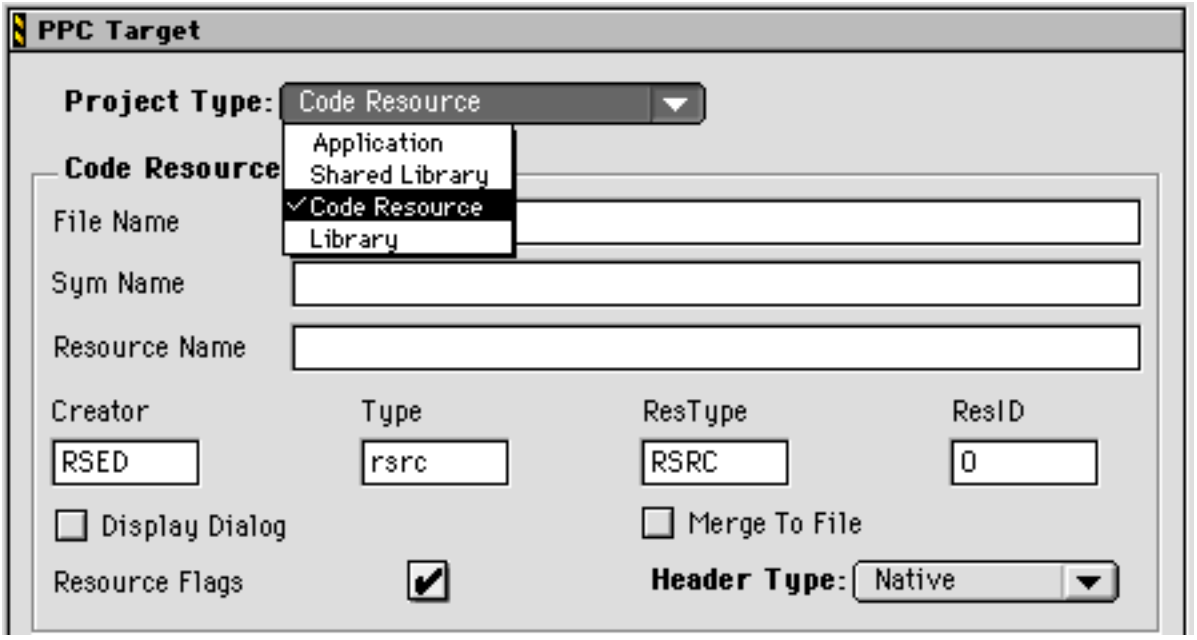
3. **Give your target a name.**

   Type a name into the Target Name field, as shown in Figure 6.5.

4. **Choose Code Resource from the Project Type menu.**

   Use the **PPC Target** panel to choose the Code Resource Project Type from the pop-up menu. This is shown in Figure 6.6.

Figure 6.6 Choose Code Resource Project Type



5. **Give your code resource a name using the File Name field in the panel shown in Figure 6.6.**

   You can also change the Creator, Type, ResType, ResID, Header Type, and other parameters at this point.

6. **Add all the needed libraries to your project.**

   For more information on selecting libraries to add to your project, see "Choosing Libraries for Code Resource Projects" on page 121.

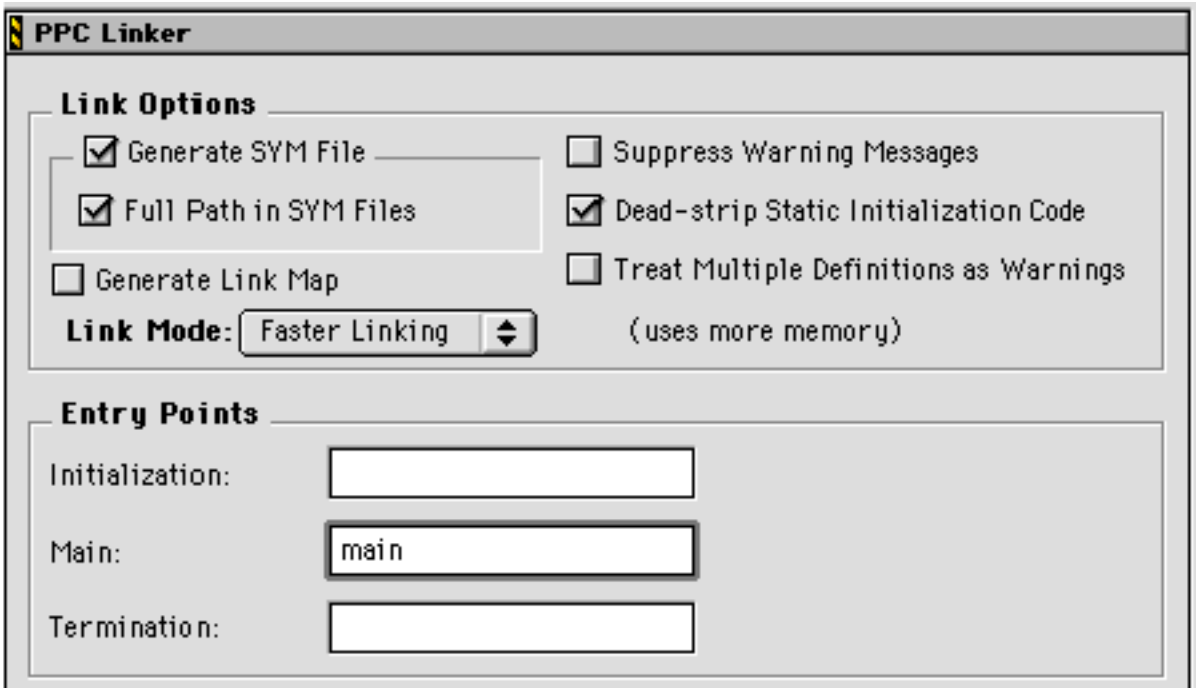
7. **Set project options.**

   You can set the options for code generation and debugging at this time. To learn how to do this, refer to "Setting Code Resource Options" on page 114.

8. **In PPC Linker panel, set the resource's main entry point.**

   By default, the name of a PowerPC application's main entry point is `__start`, defined in `MSL RuntimePPC.Lib`. Since `__start()` is for applications, you should call your main entry point to your code resource something else, such as `main`, as shown in the Main Entry Point text field of Figure 6.7.

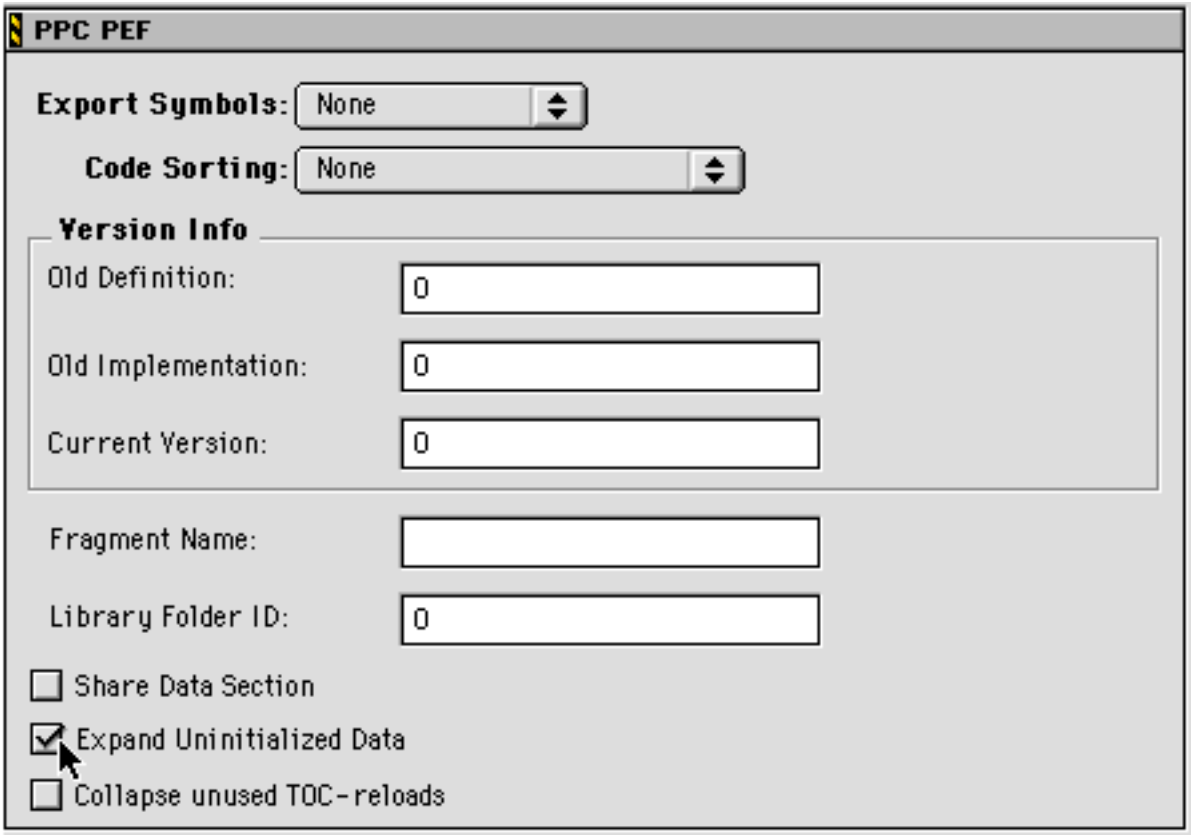
Figure 6.7 PPC Linker Options for a PowerPC Code Resource Project



9. **In PPC PEF panel, turn on the Expand Uninitialized Data option.**

   When this option is on, the linker does not pack initialized data, and allocates space for the zero-initialized portion of the data area in the fragment. It must be turned on when you're creating a PowerPC code resource, as shown in Figure 6.8.

Figure 6.8 PPC PEF Settings for a PowerPC Code Resource



10. **Configure project options.**

    For more information on how to configure the PowerPC code resource project options, refer to "Setting Code Resource Options" on page 114.

Creating 68K Single or Multi Segment Resources

When you create a 68K code resource, you must determine whether it will be a single-segment or multi-segment resource. Generally, you'll create a single-segment code resource if your resource is under 32Kbytes in size, and you'll create a multi-segment code resource if your resource is over 32K.

However, you can use small model resources over 32K if you make sure that any functions that call each other are within 32K of each other. If the functions are any further apart, CodeWarrior will generate linker error messages when it links your resource.

When you're writing a multi-segment resource, you should unload its segment with the function `UnloadA4Seg()` instead of `UnloadSeg()`. `UnloadA4Seg()` is declared in `A4Stuff.h` as this:

```
pascal void UnloadA4Seg(void *);
```

The topics discussed in this chapter include:

- Creating a Single-Segment Resource Manually
- Creating a Single-Segment Resource Automatically
- Creating a Multiple Segment Resource

Creating a Single-Segment Resource Manually

To create a single-segment code resource, configure your project so all its libraries and source files are in one segment in the project window. Then, remove all `#pragma segment` directives from your source code files and move all your files in the first group of the project window.

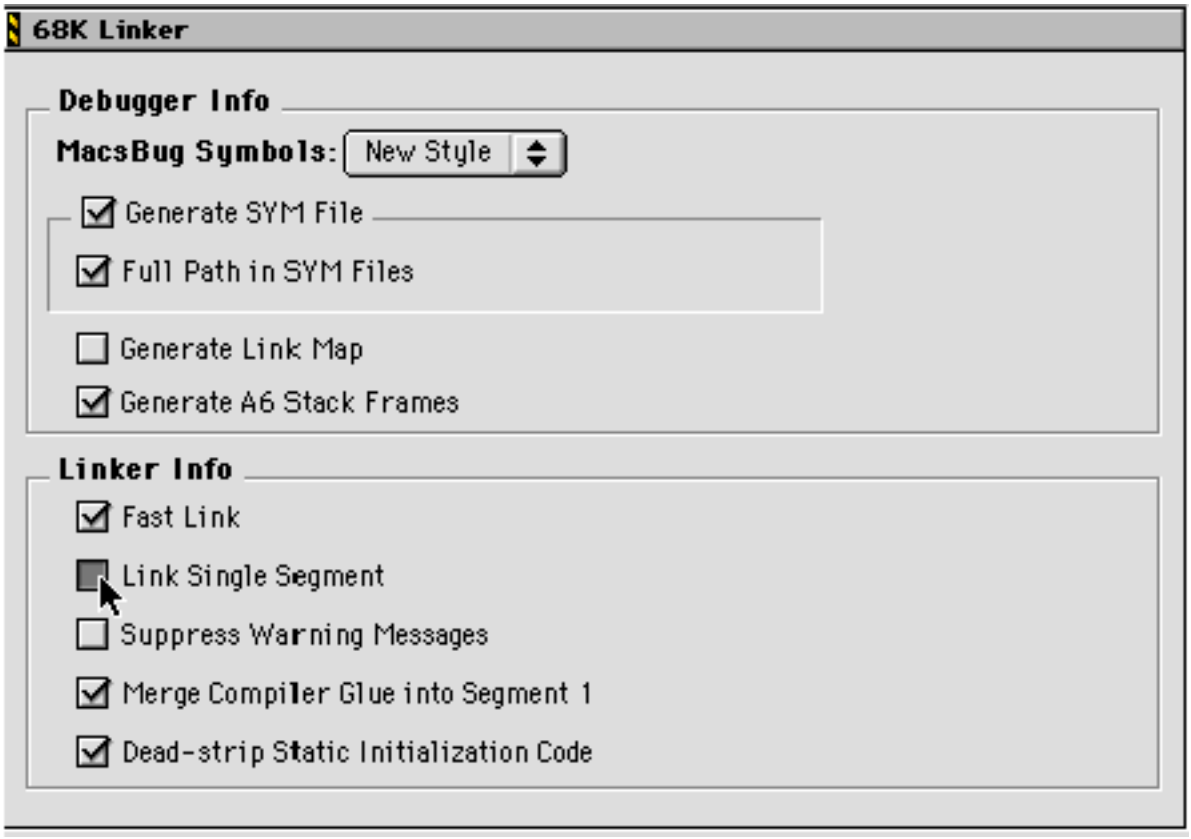
Creating a Single-Segment Resource Automatically

To create single-segment code resources automatically, you perform the following steps:

1. **Turn on the Link Single Segment option in the 68K Linker panel.**

   Refer to Figure 6.9 to see where to set this. This option causes the CodeWarrior IDE to ignore all `#pragma segment` directives and the groups in the project window. It puts all your code into one segment at link time when this option is checked.

Figure 6.9 Link Single Segment Option for 68K Linker



2. **If the resource is under 32K bytes in size, use the small code model**

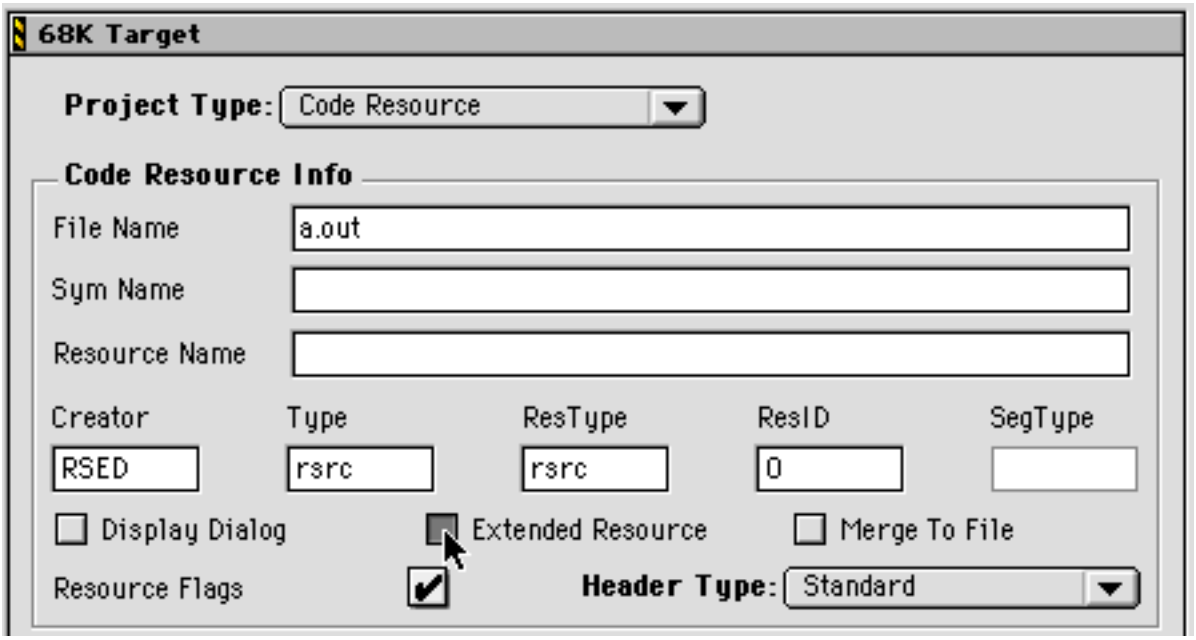
   In the **68K Processor** panel, choose Small from the Code Model pop-up menu, as shown in Figure 6.10.

Figure 6.10 Choosing the Large Code Model from 68K Processor Options

3. **If the resource is over 32K bytes in size, use the large code model and turn on the Extended Resource option.**

   In the **68K Processor** panel, choose Large from the Code Model pop-up menu, as shown in Figure 6.10. Then, in the **68K Target** panel, turn on the Extended Resource option, as shown in Figure 6.11.

Figure 6.11 Choosing Extended Resource Option from 68K Target panel



Creating a Multiple Segment Resource

1. **Separate your project into segments.**

   You can create the segments in the project window, or with the `#pragma segment` directive. For more information on pragmas, read the *C Compilers Reference* documentation on the CodeWarrior CD. For more information on creating segments in the project window, read the *IDE User Guide*.

2. **Turn on the Extended Resource option.**

   In the **68K Target** panel, turn on the Extended Resource option, as shown in Figure 6.11. This option makes sure that your resource uses the correct addressing mode.

3. **Turn off the Link Single Segment option.**

   Turn off the Link Single Segment option in the **68K Linker** panel. Refer to Figure 6.9 on page 112 to see where this setting is located. Turning this option off ensures that the linker will honor the segments that you created in Step 1.

Setting Code Resource Options

This section describes how to set the options in the code resource Target Settings panels, as shown in Figure 6.12 and Figure 6.13.

The topics discussed in this section include:

- Setting the Resource Names
- Setting the Resource Type and ID
- Setting the Resource File Type and Creator
- Setting the Resource Attributes
- Choosing Where to Save the Resource File

Figure 6.12 68K Code Resource Project Options

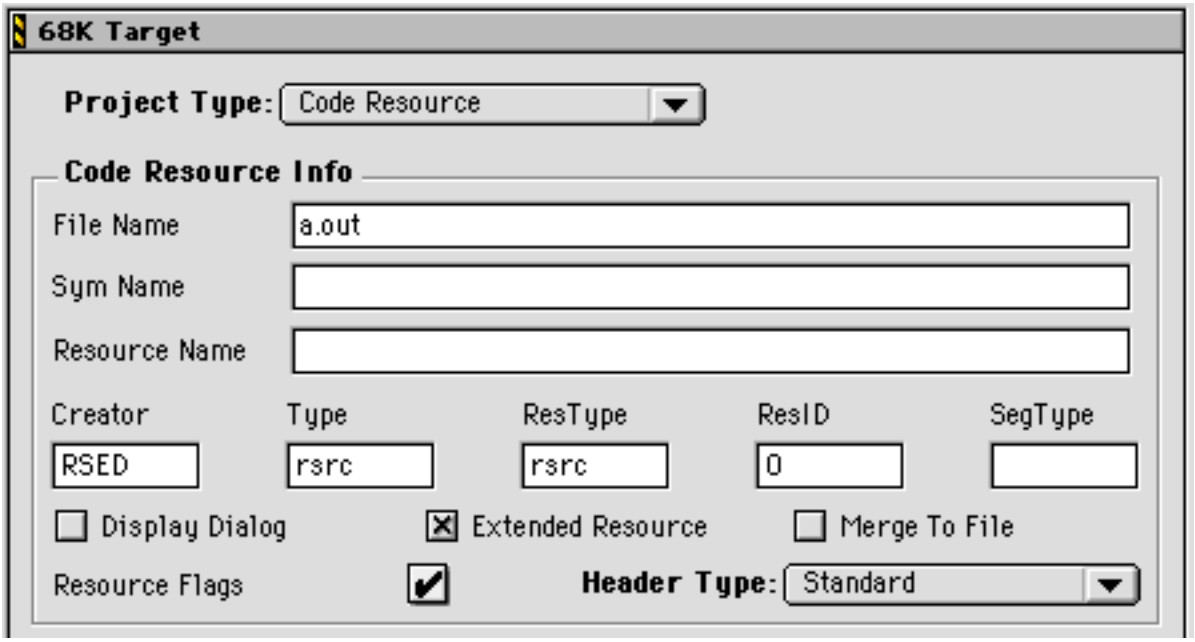
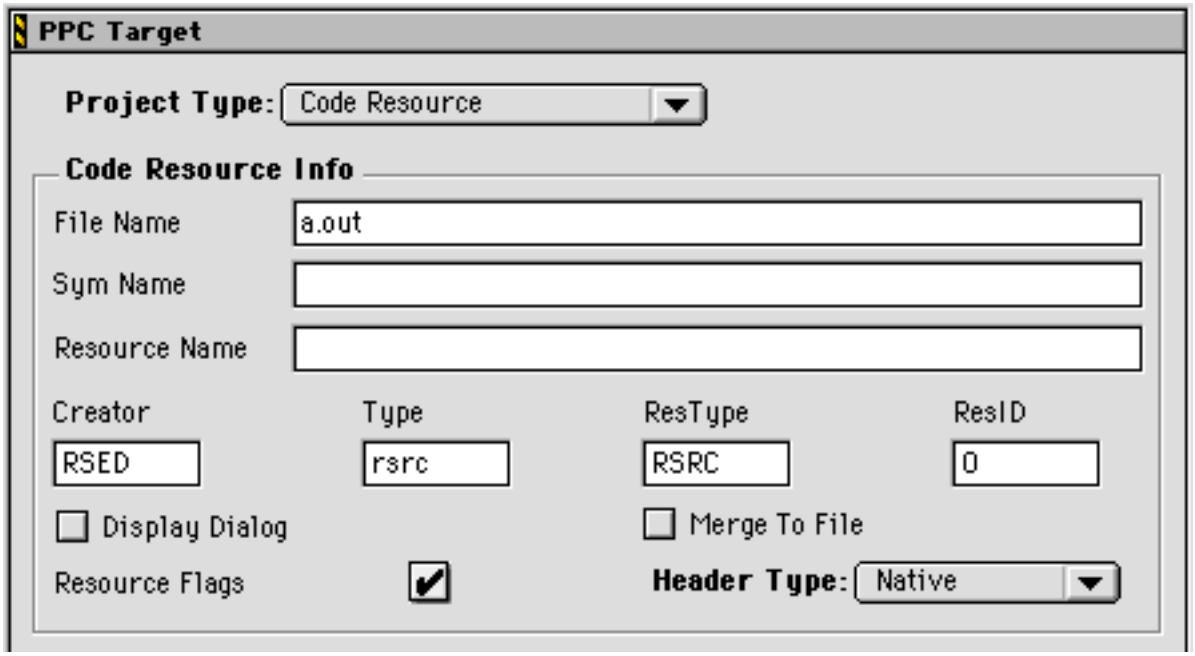


Figure 6.13 PPC Code Resource Project Options



Setting the Resource Names

When the CodeWarrior Linker creates your 68K or PPC resource, it gives the resource file the name specified in the File Name field, as shown in Figure 6.12 and 6.13. The linker also gives the resource itself the name specified in the Resource Name field on the same options panels.

NOTE: If you're creating a multi-segment resource, the CodeWarrior Linker only names the main resource, not its segments.

When the CodeWarrior Linker creates a SYM file for your resource, it gives the SYM file the name specified in the Sym Name field. CodeWarrior creates a SYM file only if you turn on the Generate SYM files option in the **68K Linker** or **PPC Linker** panels.

To learn more on the topic of configuring the generation of SYM files and debugging information, refer to "PPC Linker" on page 195 or "68K Linker" on page 291.

Setting the Resource File Type and Creator

The Creator and Type fields specify the creator and type codes for the file that contains your resource. The proper values for these fields depend on what kind of resource you're creating. The most common values are mentioned here.

Creator Field

The Creator field specifies which application opens your resource file. By default, CodeWarrior sets this field to 'RSED', which is the Creator for the ResEdit visual resource editor application.

Type Field

The Type field lets an application or the system software identify what the file contains. For example, 'INIT' is used as the Type for a system extension and 'PRER' is used for a Chooser extension.

To find out the proper values for your resource, check the documentation that describes how to create it. For example, if you are creating a Photoshop plugin, check the Adobe documentation that tells how to write Photoshop plugins.

Setting the Resource Type and ID

The ResType and ResID fields specify the resource type and ID number for your resource. If you're creating a 68K multi-segment resource, you also need to specify a SegType.

ResType

The ResType is a four-character code that identifies what kind of resource you're creating. For example, 'XCMD' is for a HyperCard XCMD,'XDEF' is for a window definition, and 'MDEF' is for a menu definition.

ResID

The ID number, along with the resource type, uniquely identifies the resource to the system software. In general, the number may be between 128 and 32,767.

SegType

In a 68K multi-segment resource, the SegType is the resource type for all the resource segments besides the main segment. The resource type for the main segment is the ResType.

Setting the Resource Attributes

Resource attributes are flags that let the Mac OS system Resource Manager know how to handle your resource. To set the flags, use the Resource Flags pop-up menu, as shown in Figure 6.14. Checked items in the pop-up menu are turned on, unchecked items are turned off. The pop-up menu may contain any number of checked items, so you can use it multiple times to toggle the attributes on or off.

Figure 6.14 Resource Attributes Pop-up Menu

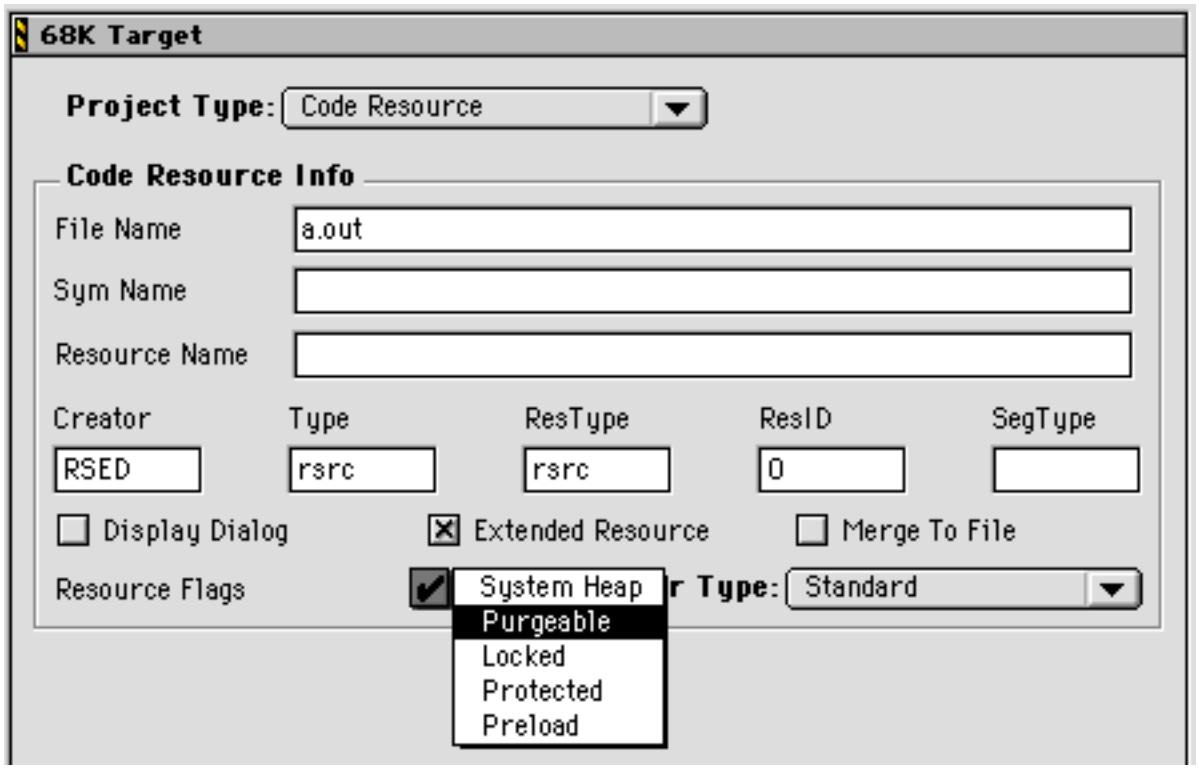


Table 6.1 describes what each resource attribute flag does. For more information on these flags, refer to the chapter called "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox,* published by Addison-Wesley.

Table 6.1 Resource attributes

Turn this attribute on…	To…
Preload	Load the resource into memory as soon as the Resource Manager opens it. When the application is launched, anything set with preloaded is loaded.
Protected	Protect the contents, name, and ID of the resource. If a resource is protected, it cannot be removed from the file to which it belongs.
Locked	Lock the resource in the heap so the system cannot move it. It gets locked into the heap's low memory address space.
Purgeable	Purge the resource from memory when the system needs to make more memory available.
System Heap	Place the resource in the system heap. If your resource is too large, it is placed in the application heap. If you turn this option off, your resource is placed in the application heap.

Choosing Where to Save the Resource File

By default, the CodeWarrior Linker saves your resource file in the same folder as your project, with the name specified in the File Name field. You save the file somewhere else with the options named Display Dialog and Merge To File, as shown in Figure 6.15. These options are discussed in Table 6.2.

Figure 6.15 Display Dialog and Merge To File Options

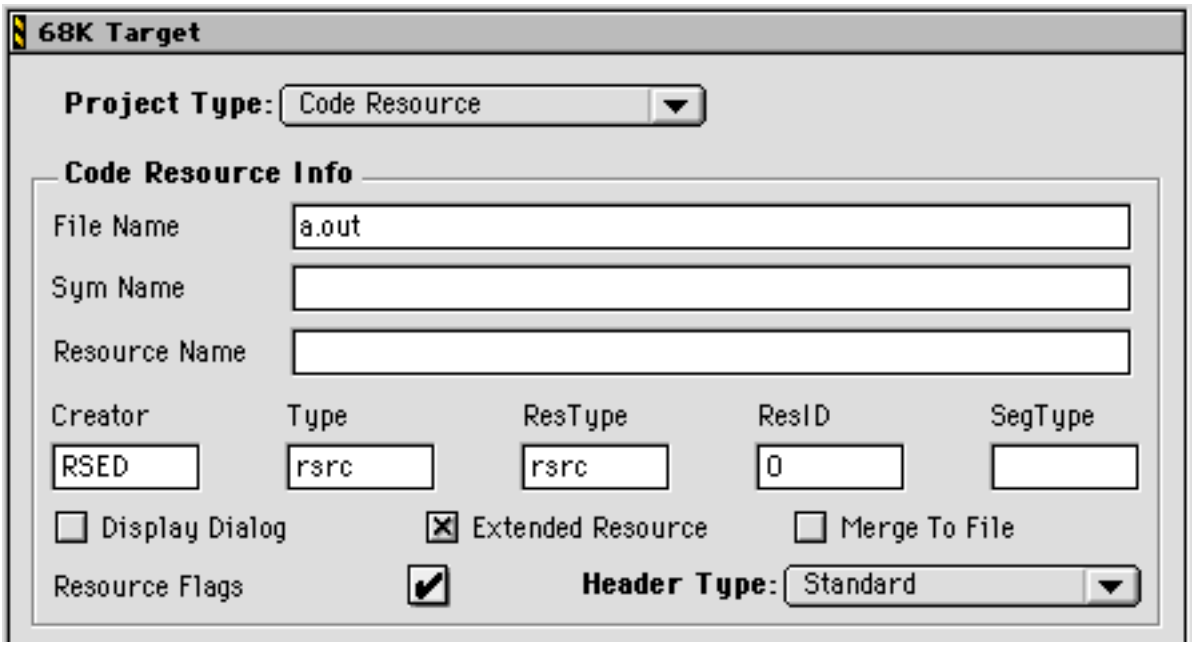


Table 6.2 Choosing where to save your resource

To save the resource here…	Do this…
In a new file in the same folder as your project	Turn off Display Dialog, Turn off Merge To File, and Enter the name in the File Name field.
In an existing file in the same folder as your project	Turn off Display Dialog, Turn on Merge To File, and Enter the name in the File Name field.
In a new file in a folder of your choosing	Turn on Display Dialog and Turn off Merge To File.
In an existing file in a folder of your choosing	Turn on Display Dialog and Turn on Merge To File.

Choosing Libraries for Code Resource Projects

This section describes which libraries to include in 68K and Power Macintosh code resource projects.

The topics in this section include:

- Libraries For 68K Code Resource Projects
- Libraries For PowerPC Code Resource Projects

Libraries For 68K Code Resource Projects

The libraries that should be added to your project for creating 68K code resources are shown in Table 6.3.

If the library's name in Table 6.3 contains *xx*, you need to choose a library based on some **Target Settings** panel options. To choose the correct library for the *xx* in your project, refer to the naming conventions discussed in "Library Naming Conventions" on page 421.

Once you have added the correct libraries to your project, you need to make sure that the correct Target Settings are specified correctly for your project. For example, if you include a library with "2i" in the name, you must not be using 4-byte integers in the Target Settings. To learn how to configure the Target Settings, refer to "Target Settings for 68K" on page 249.

Table 6.3 Choosing libraries to add to an 68K code resource project

If the project…	Add this…
Contains any C++ code	`MSL Runtime68K.A4.Lib`
Uses custom startup code	MSL CustomCRStartup.Lib
Calls any Macintosh Toolbox routines	`MacOS.lib`
Is a HyperCard XCMD or XFCN	`HyperXLib.o`

If the project…	Add this…
Calls any math routines	`MathLib68K (xx).A4.Lib`
Calls any ANSI C standard library routines	`MSL C.68K.xx.A4.Lib` libraries, such as `MSL C.68K (2i).A4.Lib` Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Contains any C++ code and calls any ANSI C standard library routines	`MSL C++.68K.xx.A4.Lib` libraries, like `MSL C++.68K (2i).A4.Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.

The names of many of these libraries include "A4" in the name. These libraries contain global variables, and use the 68K A4 register to reference them instead of A5. Be sure you include the A4 version of these libraries.

NOTE: Whenever you must access Global variables or functions, you must include the A4 libraries.

Libraries For PowerPC Code Resource Projects

Table 6.4 lists the CodeWarrior libraries you might need to add to your PowerPC code resource projects.

NOTE: MSL RuntimePPC.Lib should be included in Power Macintosh code resource projects, especially if they use C++ code.

Table 6.4 Choosing libraries for a PowerPC Mac OS resource project

If the project…	Add this…
Always	`InterfaceLib` and `MSL RuntimePPC.Lib`
Calls any math routines	`MathLib`
Calls any ANSI C standard library routines and uses the SIOUX console window	`MSL C.PPC.Lib`
Calls ANSI C standard library routines and contains C++	`MSL C.PPC.Lib` and `MSL RuntimePPC.Lib`
Uses the C++ complex number class	`MSL C.PPC.Lib` and `MSL RuntimePPC.Lib`

Writing Code Resources

Writing a code resource can require more work than writing an application. This section points to some issues to watch out for when writing a code resource with CodeWarrior. It describes how to set up global variables for 68K code resources, creating custom headers for 68K code resources, and calling functions in PowerPC code resources.

The topics in this section include:

- Setting Up Globals in 68K Code Resources
- Creating Custom Headers for 68K Code Resources
- Calling PowerPC Code Resources

Setting Up Globals in 68K Code Resources

Using CodeWarrior to create code resources, you can allow code resources use global data. To avoid any conflict with the running application, the resource accesses its globals referenced from the 68K A4 register, instead of A5. In a desk accessory or device driver built with the Extended Code Resource turned on, the CodeWarrior

linker appends your globals to the second segment. In any other code resource, your globals are appended to the first segment.

If you're writing a desk accessory or device driver, CodeWarrior sets up A4 for you. If you're writing any other kind of resource, you must set up A4 yourself. Call `EnterCodeResource()` when you first enter `main()` and `ExitCodeResource()` just before you leave `main()`. These functions are declared in `A4Stuff.h`. The example shown in Listing 6.1 shows how these functions should be used.

Listing 6.1 Setting up A4 for a code resource

```
#include <A4Stuff.h>
void main(void)
{
  EnterCodeResource();
  /*
   * Your program goes here.
   */
  ExitCodeResource();
}
```

`A4Stuff.h` and `SetUpA4.h` provide functions that set `A4` and are useful when you can't be sure of `A4`'s value. For an example of these routines, refer to the `Multi-Seg CR w/ Callback` example on your CodeWarrior Reference CD.

Creating Custom Headers for 68K Code Resources

A code resource header, the first few bytes in the resource, tells the system software what your resource contains. If you create a 68K code resource with a standard header and the Extended Code Resource option turned off, the CodeWarrior linker uses Apple's suggested format for a header, as shown in Listing 6.2.

Listing 6.2 Standard header for a code resource

```
  bra.s __Startupcode__
  dc.w  0
  dc.l  Resource Type     // Contents of ResType field
  dc.w  Resource ID       // Contents of ResID field
  dc.w  Version           // Always 0

    __Startupcode__:
  jmp main
```

If you create a 68K code resource with a custom header, you must do two things:

- Create a function named `__Startup__()` that contains the header. This function must be the very first piece of executable code in the first segment of your code. Be sure that all header files, including `SetupA4.h`, are included after it. This is necessary because `SetupA4.h` contains code. Declare `__Startup__()` like this:

  ```
  asm void __Startup__(void)
  ```

- Add the library `MSL CustomCRStartup.Lib` to the first segment of your project within 32K bytes of `__Startup__()`.

The following code listings are C versions of multi-segment and single segment headers. For more examples, see the `Code Resource Examples` on the CodeWarrior Reference CD.

Listing 6.3 shows a custom header for a single-segment resource.

Listing 6.3 Custom header for a single-segment code resource

```
//File:Startup.c
// Note: this must be the first file in your
// project !
// (single segment code resource version)
extern void main(void);  // This is the user main()
                         // called by __Startup__()
extern void __Startup__(void);// Resource/DRVR
                              // startup code
```

```
asm void __Startup__(void)
    // Note: this must be the first code in
    // this file !
{
    bra.s start    // 0 standard macintosh code
                   //   resource header
    dc.w0          // 2
    dc.l'WDEF'     // 4 code resource type
    dc.w123        // 8 code resource id
    dc.w0          // 10 code resource version no.

  start:
    jmp main       // direct jump to main (no
                   // special setup is needed)
}
```

Listing 6.4 shows a custom header for a multi-segment resource.

Listing 6.4 Custom header for multi-segment resource

```
//File:Startup.c
//    (Multi-segment code resource version)
//
//    This must be the first file in your project!
//

#pragma near_code
    // All code references must be
    // 16-bit PC relative
#pragma far_data off
    // All data references must be
    // 16-bit A4 relative

extern void main(void);
    // This is the user main()
    // that __Startup__() calls
extern void __Startup__(void); /
    // Code resource/DRVR startup code
extern void __SetupMainRsrc__(void *:__D0);
    // Relocation setup function
```

```
extern long SetCurrentA4(void);
    // Setup the A4 data ptr
    // (and return old A4 in D0)

      // This must be the first code in the file !
asm void __Startup__(void)
{
// This will be the start of your code resource.
// The code below is only an example. You can
// change this code, however you have to make
// sure that function "__SetupMainRsrc__()" is
// called with a setup A4 register before any
// function or variable is accessed that is not
// referenced by a 16-bit A4/PC relative offset.

  start:
    bra.s setup// 0 standard macintosh code
               //   resource header
    dc.w 0    // 2

    dc.l 'WDEF'// 4code resource type
    dc.w 123  // 8code resource id
    dc.w 0    // 10 code resource version num

  setup:
    movem.l d0-d2/a0/a1/a4,-(sp) // Save registers
    jsr     SetCurrentA4     // Setup A4 for
                             // addressing globals

    lea     start,a0         // Pass a pointer to
    move.l  a0,d0            // the start of this
                             // function in D0
    _StripAddress
    jsr     __SetupMainRsrc__ // Relocate main
                               // resource

    movem.l (sp)+,d0-d2/a0/a1/a4// Restore
                                // registers
```

```
    jmp     main            // And jump to main
}
```

Calling PowerPC Code Resources

In the past, when you called a routine in a 68K code resource, you could just load the resource and call the routine. But now, when you call a routine in a PowerPC code resource, you must use the routine `CallUniversalProc`(). This creates a problem if you're creating a fat application. You must use macros to make sure that the 68K code calls code resource routines one way and that the PowerPC code calls code resource routines another way. In the example below, the macro `CallMainProc` helps you avoid that problem.

Listing 6.5 Calling a code resource

```
typedef void(*MainProcPtr)(void);

enum {
  uppMainProcInfo = kThinkCStackBased
};

#if USESROUTINEDESCRIPTORS
typedef UniversalProcPtr MainProcUPP;
#define CallMainProc(userRoutine)\

CallUniversalProc((UniversalProcPtr)(userRoutine),
                  uppMainProcInfo)
#else
typedef MainProcPtr MainProcUPP;
#define CallMainProc(userRoutine)\
    (*(userRoutine))()
#endif

main()
{
  Handle    resource;
  MainProcUPPmyProcPtr;
```

```
  resource = Get1Resource('MWCW', 128);
  HLockHi (resource);
  myProcPtr = (MainProcUPP)*resource;
  CallMainProc(myProcPtr);
}
```

The macro `CallMainProc` takes care of several things:

- It fills in the parameters to `CallUniversalProc()` for you.
- It handles a fat application properly. The `#if USESROUTINEDESCRIPTORS` directive checks whether you're creating a 68K or PowerPC application and defines the correct version of `CallMainProc`.

You'll find more interesting PowerPC declarations in the header file named `MixedMode.h`, installed in the `Universal Headers` folder in your CodeWarrior installation folder.

7

Creating MPW Tool Projects for Mac OS

An MPW tool is a program that you can run under MPW (Macintosh Programmers Workshop). MPW provides a UNIX-like console user interface.

This section describes the following topics:

- Creating an MPW Tool Project
- Choosing Libraries for an MPW Tool Project
- Writing an MPW Tool
- Running the MPW Tool

To learn more about MPW tools, refer to *Building Programs in MPW*, included on the CodeWarrior Reference CD.

Creating an MPW Tool Project

This section describes how to create a 68K or PowerPC MPW tool project.

The topics in this section include:

- Creating a 68K MPW Tool Project
- Creating a PowerPC MPW Tool Project
- Creating a Fat MPW tool

Creating a 68K MPW Tool Project

The steps for creating a 68K MPW tool are:

1. **Create an empty project.**

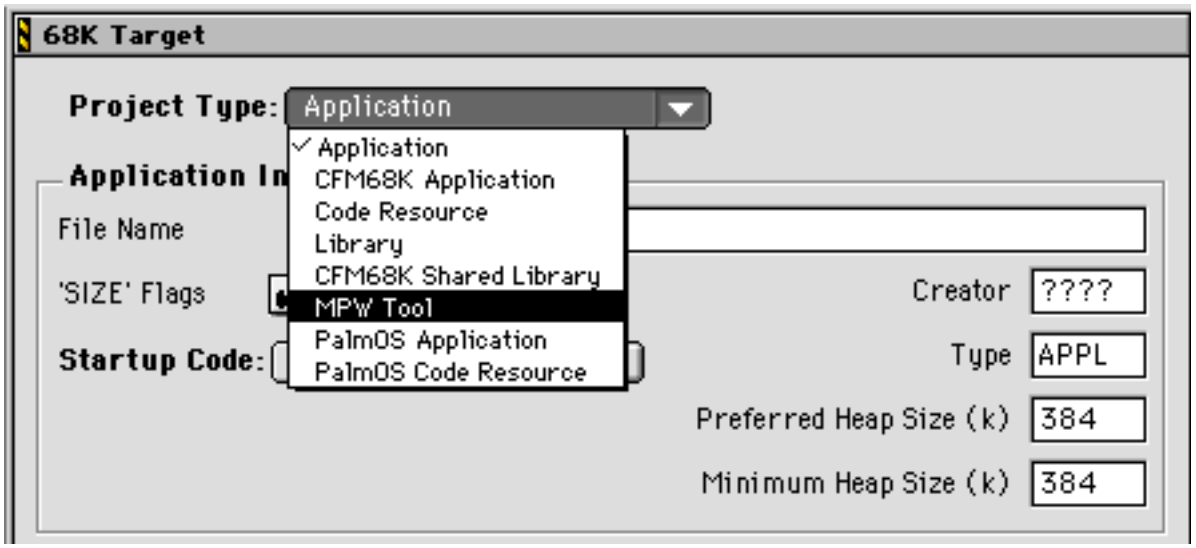
   Use the New Project command under the File Menu, and select Empty Project from the New Project Stationery dialog.

2. **In the 68K Target panel, choose Mac OS 68K Linker from the Linker pop-up menu.**

3. **In the 68K Target panel, choose MPW Tool from the Project Type menu.**

   Figure 7.1 shows how to do this.

Figure 7.1 Choosing MPW Tool as the Project Type

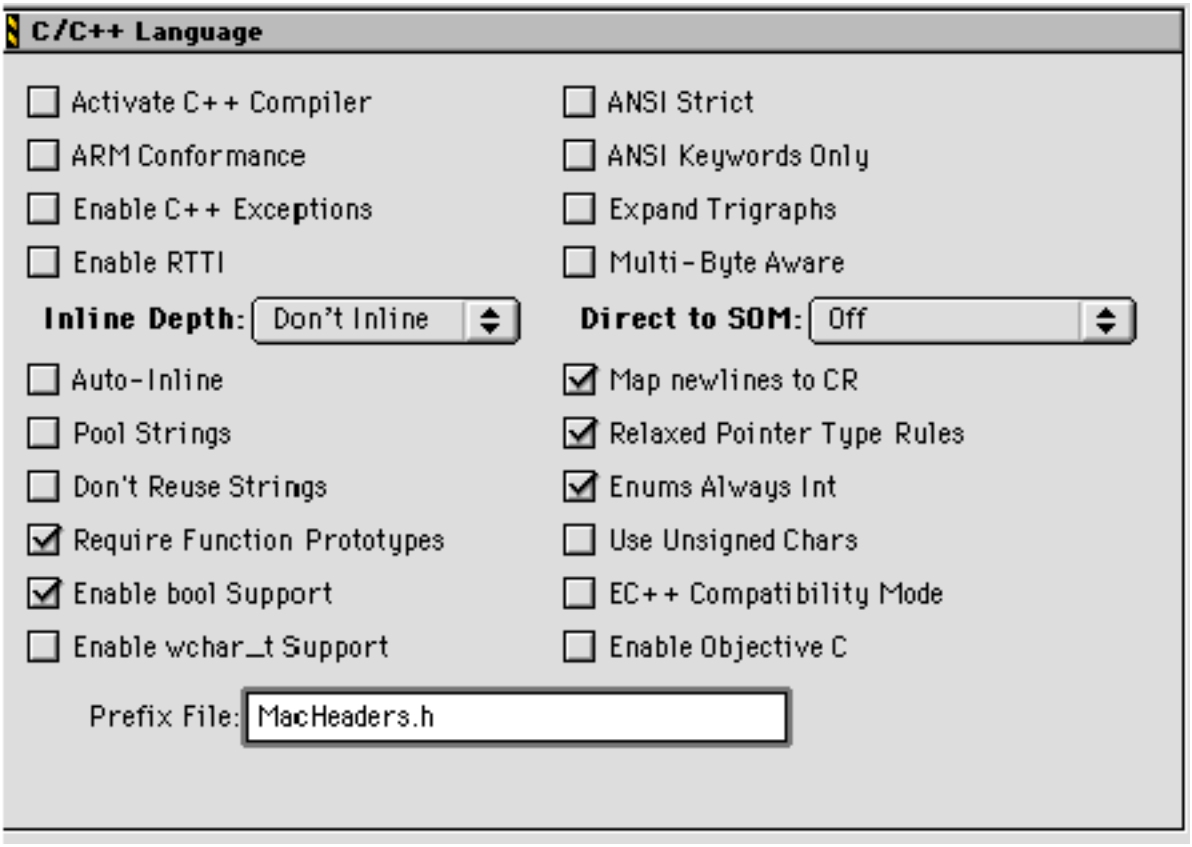


4. **Enter the name of the tool in the File Name field.**

5. **In the C/C++ Language panel, choose MPW C or MPW C++ from the Source Model menu.**

   Figure 7.2 shows how to do this.

Figure 7.2 Choosing MPW C for the Project's Language



MPW Shell expects its tools to be compiled with options that are compatible with MPW C. These options need to be enabled:

- MPW Newlines
- MPW Pointer Type Rules
- Enums Always Int
- Require Function Prototypes
- Enable bool Support

Additionally, if you're using C++, you need to enable:

- Activate C++ Compiler
- Enable C++ Exceptions
- Enable RTTI

6. **In the Access Paths panel, change the folders in the System Paths pane.**

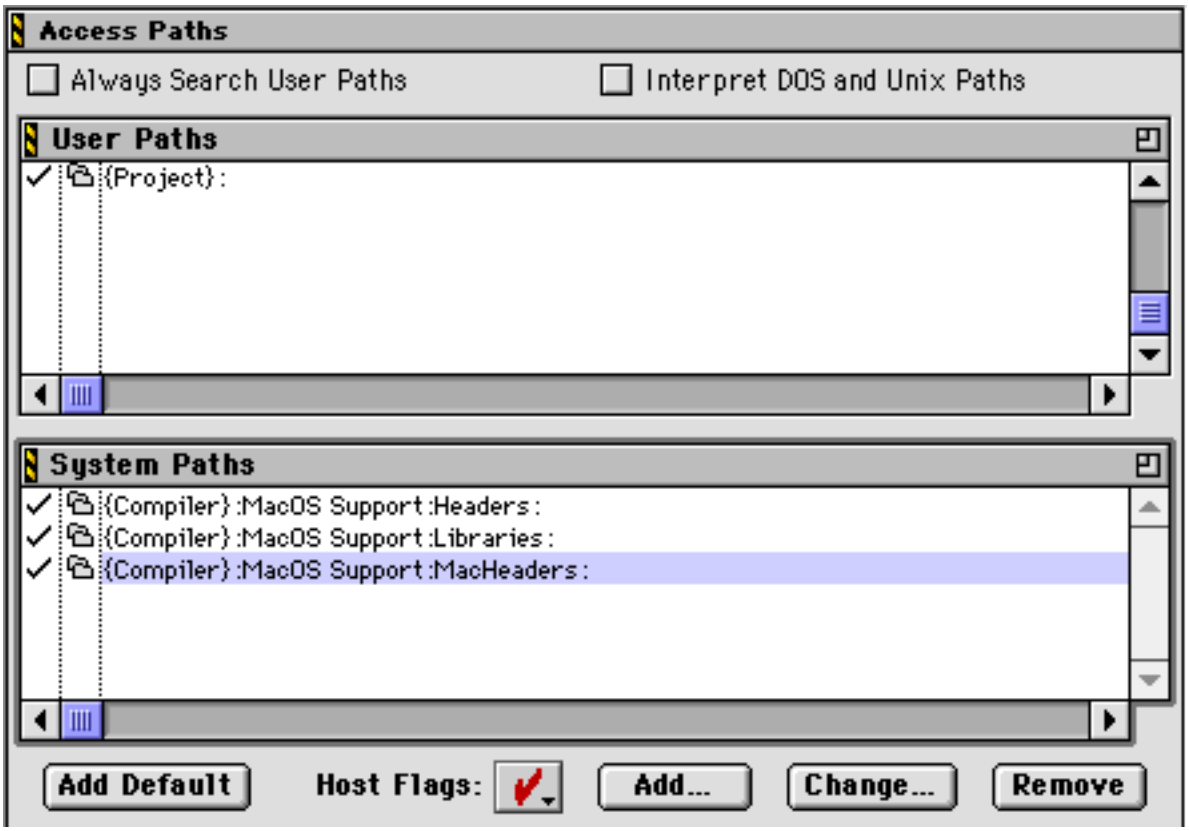
   Remove the {Compiler *f*} from the System Paths pane, and add these folders:

   - Headers
   - Libraries
   - MacHeaders

   NOTE: You must add the folders in the order listed!

   The **Access Paths** panel should now look something like that shown in Figure 7.3.

Figure 7.3 Access Paths for 68K MPW Tools



7. **In the 68K Processor panel, turn on the 4-byte Ints option**

   The MPW Shell uses 4-byte integers.

8. **Add all the needed libraries to the project.**

   For more information on choosing libraries, refer to "Choosing Libraries for an MPW Tool Project" on page 147.

9. **If any source file includes fcntl.h or ioctl.h, make sure you use Apple's version instead of Metrowerks's.**

   The header files fcntl.h and ioctl.h declare functions that the Apple and Metrowerks libraries implement in incompatible ways. When you create MPW tools, you must use Apple's versions.

   In the source file, edit the #include statements so they use quotes ("") instead of brackets (<>). For example, change

```
#include <fcntl.h>
```

to

```
#include "fcntl.h"
```

Then make copies or aliases of the files `fcntl.h` and `ioctl.h` in the CIncludes folder, and put the copies or aliases in your project's folder. If you use the file fcntl.h, also make a copy or alias of the file Seekdefs.h in the CIncludes folder and put it in the ANSI Headers folder in the Headers folder in the MacOS Support folder in your Metrowerks CodeWarrior folder. The CIncludes folder is in the Interfaces folder in the CodeWarrior MPW folder.

NOTE: Apple's version of `fcntl.h` is incompatible with Metrowerks.

Creating a PowerPC MPW Tool Project

These are the steps to follow to create a PowerPC MPW Tool.

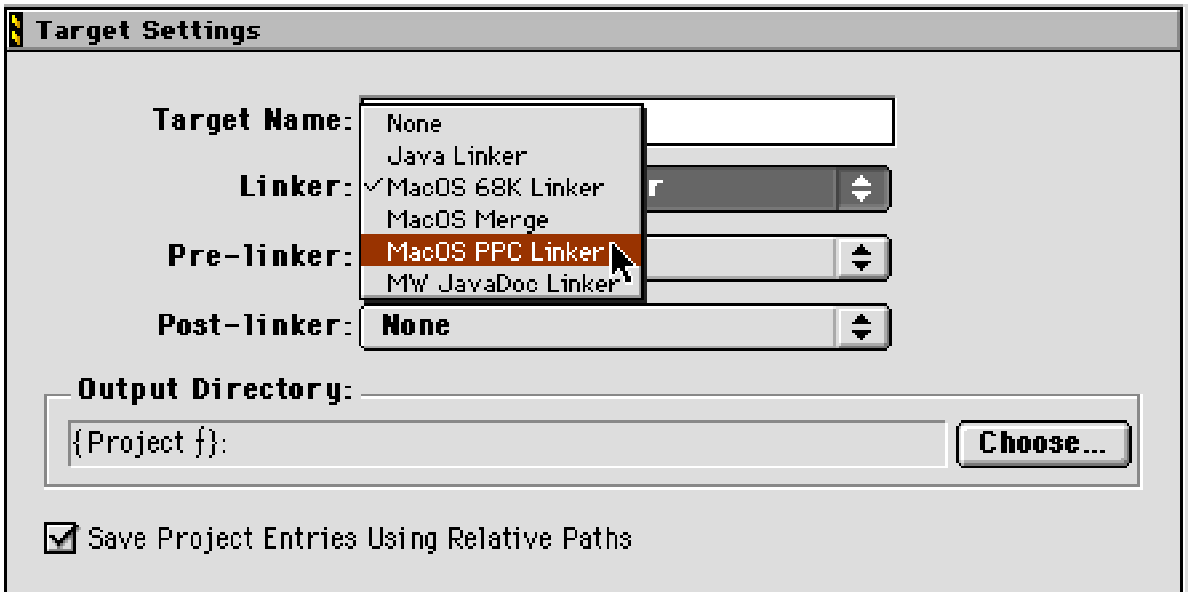
1. **Create an empty PowerPC project.**

   Use the New Project command under the File Menu, and select Empty Project from the New Project Stationery dialog.

2. **In the Target Settings panel, choose Mac OS PPC Linker from the Linker pop-up menu.**

   Figure 7.4 shows how to do this.

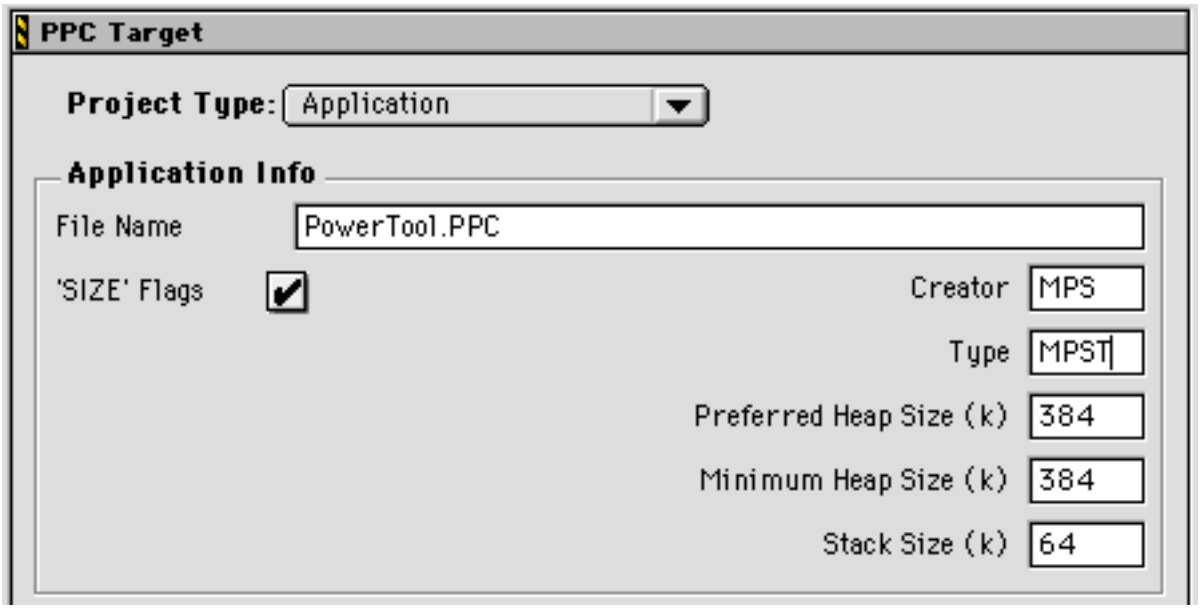
Figure 7.4 Choosing the Mac OS PPC Linker



3. **In the PPC Target panel, choose Application from the Project Type menu.**
4. **Enter the name of the tool in the File Name field.**
5. **Enter the Creator and File Type codes.**

   Enter 'MPS ' (note the space) as the Creator and 'MPST' as the Type. The **PPC Target** panel should look like Figure 7.5.

Figure 7.5 PowerPC MPW Tool Target Settings



6. **In the C/C++ Language panel, set options for MPW C or MPW C++ compatibility.**

   MPW Shell expects its tools to be compiled with options that are compatible with MPW C. These options need to be enabled:

   - MPW Newlines
   - MPW Pointer Type Rules
   - Enums Always Int
   - Require Function Prototypes
   - Enable bool Support

   Additionally, if you're using C++, you need to enable:

   - Activate C++ Compiler
   - Enable C++ Exceptions
   - Enable RTTI

7. **In the Access Paths panel, change the folders in the System Paths pane.**

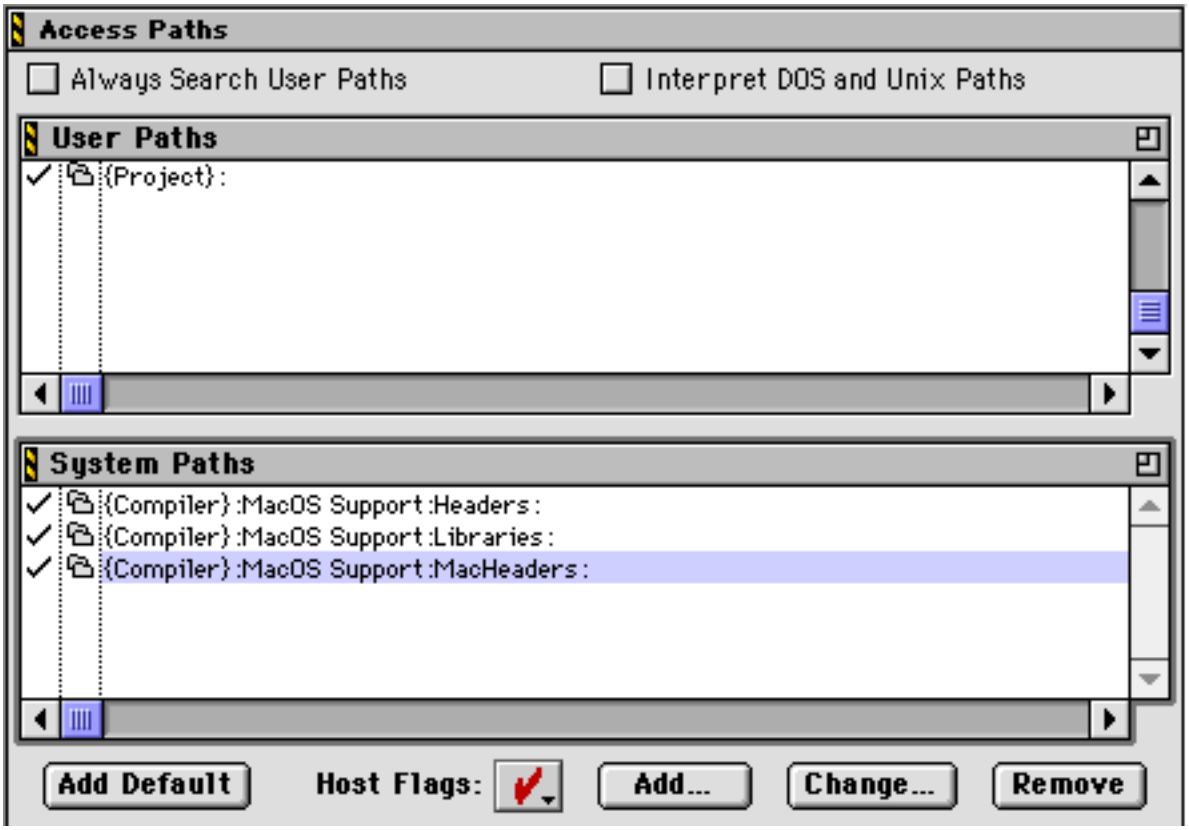
   Remove the {Compiler *f*} from the System Paths pane, and add these folders:

- Headers
- Libraries

NOTE: You must add the folders in the order listed!

The System Paths pane of the **Access Paths** panel should look something like that shown in Figure 7.6.

Figure 7.6 Access Paths for PowerPC MPW Tools



8. **Add all the needed libraries to the project.**

   For more information on choosing libraries, refer to "Choosing Libraries for an MPW Tool Project" on page 147.

9. **If any source file includes fcntl.h or ioctl.h, make sure you use Apple's version instead of Metrowerks's.**

   The header files fcntl.h and ioctl.h declare functions that the Apple and Metrowerks ANSI libraries implement in incompatible ways. When you create MPW tools, you must use Apple's versions.

   In the source file, edit the #include statements so they use quotes (“”) instead of brackets (<>). For example, change

   ```
   #include <fcntl.h>
   ```

   to

   ```
   #include "fcntl.h"
   ```

   Then make copies or aliases of the files fcntl.h and ioctl.h in the CIncludes folder, and put the copies or aliases in your project's folder. If you use the file fcntl.h, also make a copy or alias of the file Seekdefs.h in the CIncludes folder and put it in the ANSI Headers folder in the Headers folder in the MacOS Support folder in your CodeWarrior folder. The CIncludes folder is in the Interfaces folder in the CodeWarrior MPW folder.

NOTE: Apple's version of fcntl.h is incompatible with Metrowerks.

Creating a Fat MPW tool

The following procedure describes the easiest way to create a Fat MPW tool.

NOTE: This procedure could not be confirmed at press time. Please check the release notes for any late-breaking information on this topic.

There is one other method of creating Fat projects. The procedure listed in “Creating Fat Applications with Multiple Targets” on page 40 is not exactly the same as the alternate procedure you would use for creating Fat MPW Tools. However, this method illustrates some of the advanced features of the CodeWarrior IDE and is worth read-

ing. The method discussed here is designed to quickly get Fat MPW tools building.

NOTE: Only MPW Shell 3.4d1 and later or ToolServer 3.4d1 and later will use the PowerPC code in a Fat MPW tool. Older versions of MPW running on Mac OS PowerPC computers will execute the 68K code in emulation.

1. **Create two MPW tool projects: a 68K project and a PowerPC project.**

   Before you can create a fat MPW Tool, you must create working 68K and PowerPC tools.

   Create each of the two projects from scratch. Don't try to duplicate one project and convert it, since this can cause problems. Refer to

   Use the same source files and resource files in each project. Only the libraries should be different. If you need to use different source code for the PowerPC and 68K tools, use code of the form:

```
#if defined(powerc)
/* PowerPC code goes here */
#else
/* 68K code goes here */
#endif
```

   to check which CPU you're compiling for.

2. **Test and debug the PowerPC and 68K MPW tool projects.**

   Both MPW Tools must be working properly before you can create a Fat tool.

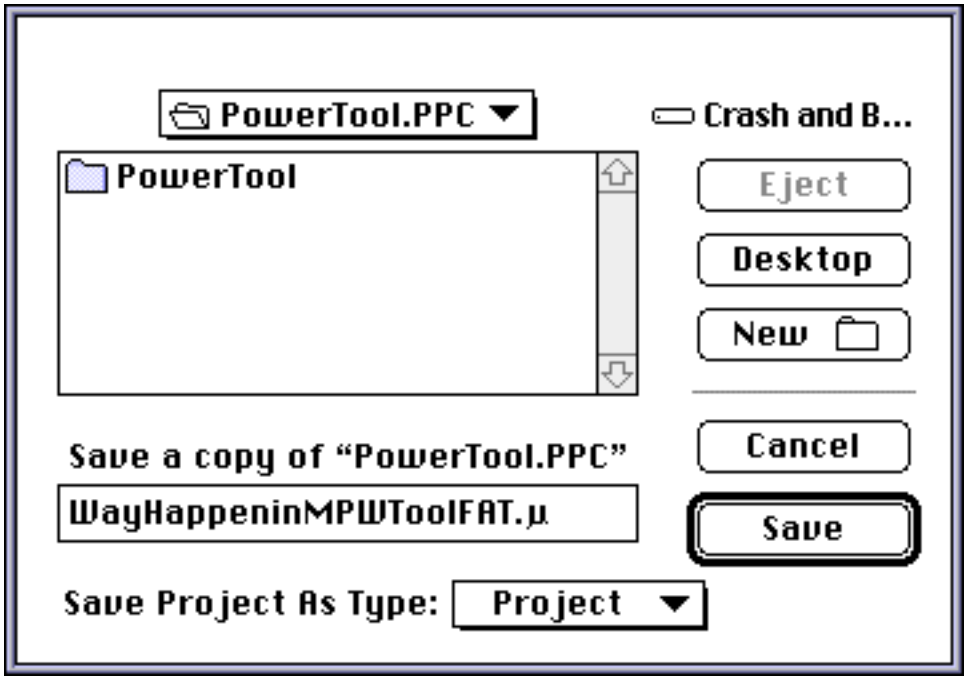
3. **Create a duplicate of your PowerPC Mac OS project.**

   Make a duplicate of the PowerPC MPW Tool project file. If you are developing your code using a Mac OS computer, you can use the **Duplicate** command in the Mac OS Finder, or the Save A Copy As command in the File Menu. If you are developing your code on a

Windows computer, you can use the Save A Copy As command in the File Menu.

Rename the duplicated project to something descriptive. For example, use `WayHappeninMPWToolFAT.µ` on the Mac OS, or `WayHappeninMPWToolFAT`.mcp on the Windows host platform. Figure 7.7 shows the dialog box that appears when you choose the Save A Copy As menu command.

Figure 7.7 Saving a Duplicate of Your Fat Project File



4. **Add the 68K project file (as a subproject) to the Fat application project.**

   Click in your open Fat project file window to make the window active, and then use the Add Files command in the Project Menu to add the 68K project file to the project. This makes the 68K project file a subproject of the Fat project. When the Fat project is built, the 68K project will automatically be updated.

   Figure 7.8 shows the dialog that appears when you use the Add Files menu command for selecting a file to add. Click on the 68K

project file you want to add, then click the Add button. Then click the Done button.

Figure 7.8 Selecting a 68K Project File to Add to the Fat Project

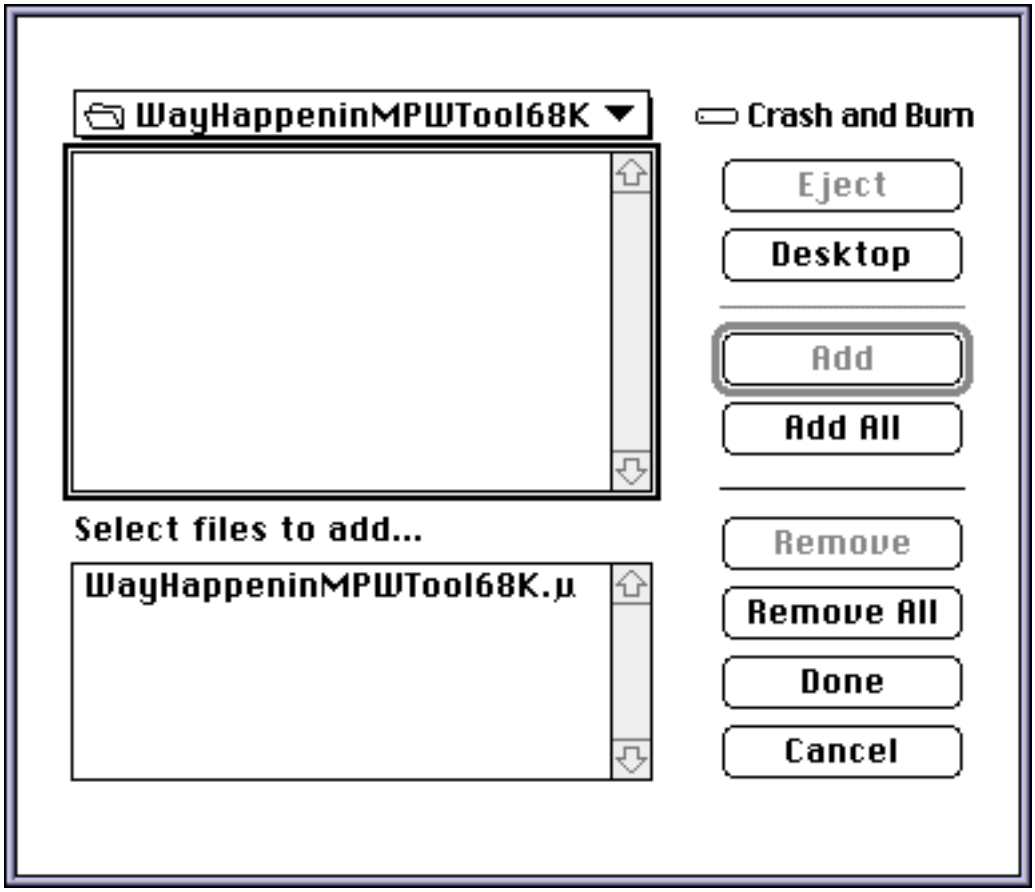
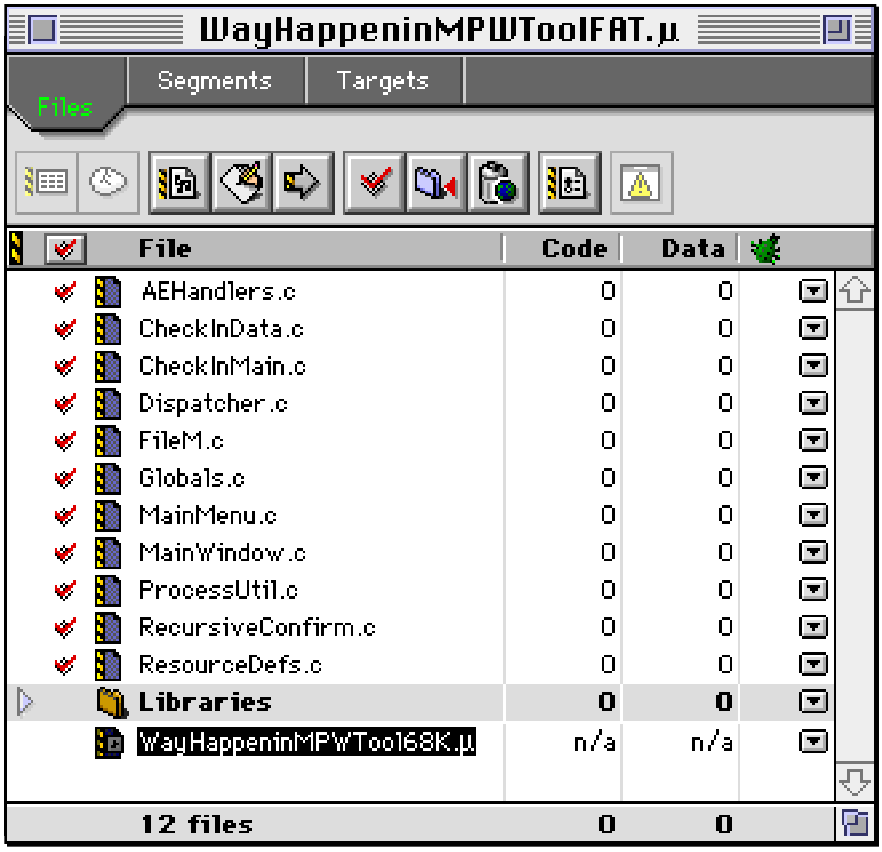


Figure 7.9 shows the Fat project file with the 68K project file added to it.

Figure 7.9 Fat Project File with 68K Subproject File



5. **Remove the resource files from the Fat project file.**

   This step is optional. When you do this step, you will minimize the number of warnings you receive at link time for the Fat project.

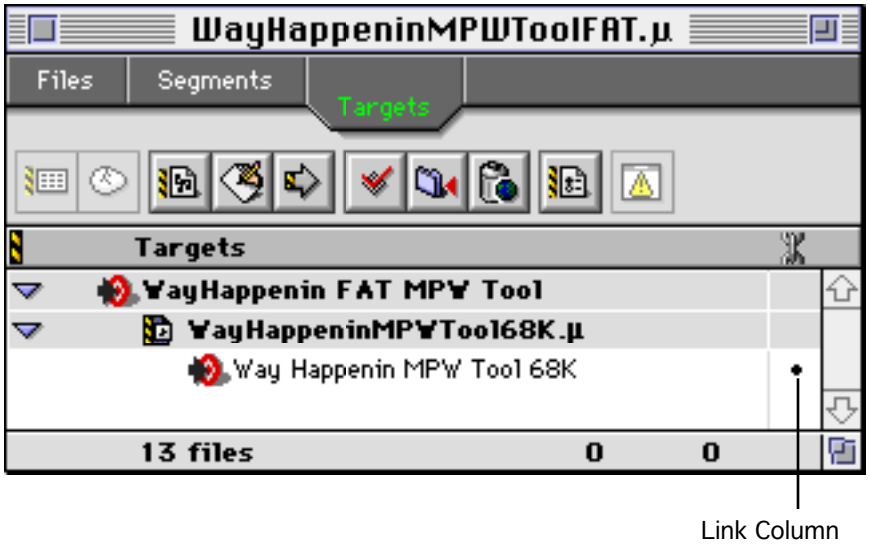
   Remove all resource files from your Fat project file, using the Remove Selected Items menu command in the Project Menu. The reason you do this is to keep duplicate resources from being used at link time, which generates link warnings.

   You aren't required to remove the resource files from the Fat project, this step is optional. But the CodeWarrior IDE creates duplicate resource warnings if you don't. You can safely ignore the warnings if they don't bother you.

6. **Configure linking for the 68K application.**

   Click in the Link Column of the Targets view of the Project window, in the same row as the 68K target, as shown in Figure 7.10. When you click there, a small dot appears in the column.

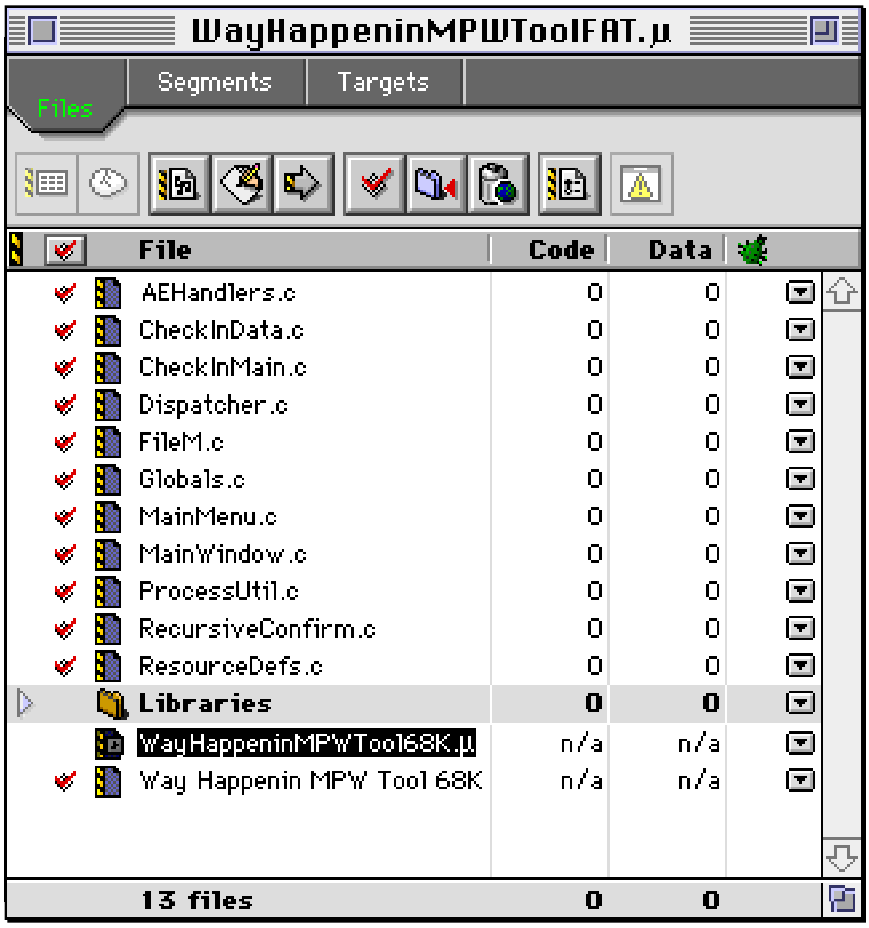
Figure 7.10 Configuring the Linking of 68K Code into the Fat MPW Tool



This causes the 68K target output file to be linked into the Fat target's output file. So now, when the Fat target is built, the 68K subproject is built first. Then the output of the 68K target is linked into the Fat target.

Notice that when you click on the Link Column, the Files view of the Project window changes, as shown in Figure 7.11. The output file of the 68K target has now been added to the project as a file.

Figure 7.11 Including the 68K File in the Project Window Files View



7. **Build the Fat project.**

   Use the **Make** command on the Project Menu to build and link your Fat project. CodeWarrior creates a new Power PC MPW Tool and copies the code resources, and the project resources, from the 68K application into the Fat application.

8. **In the File Mappings panel, add a file type entry that lets you add MPW Tools to the project.**

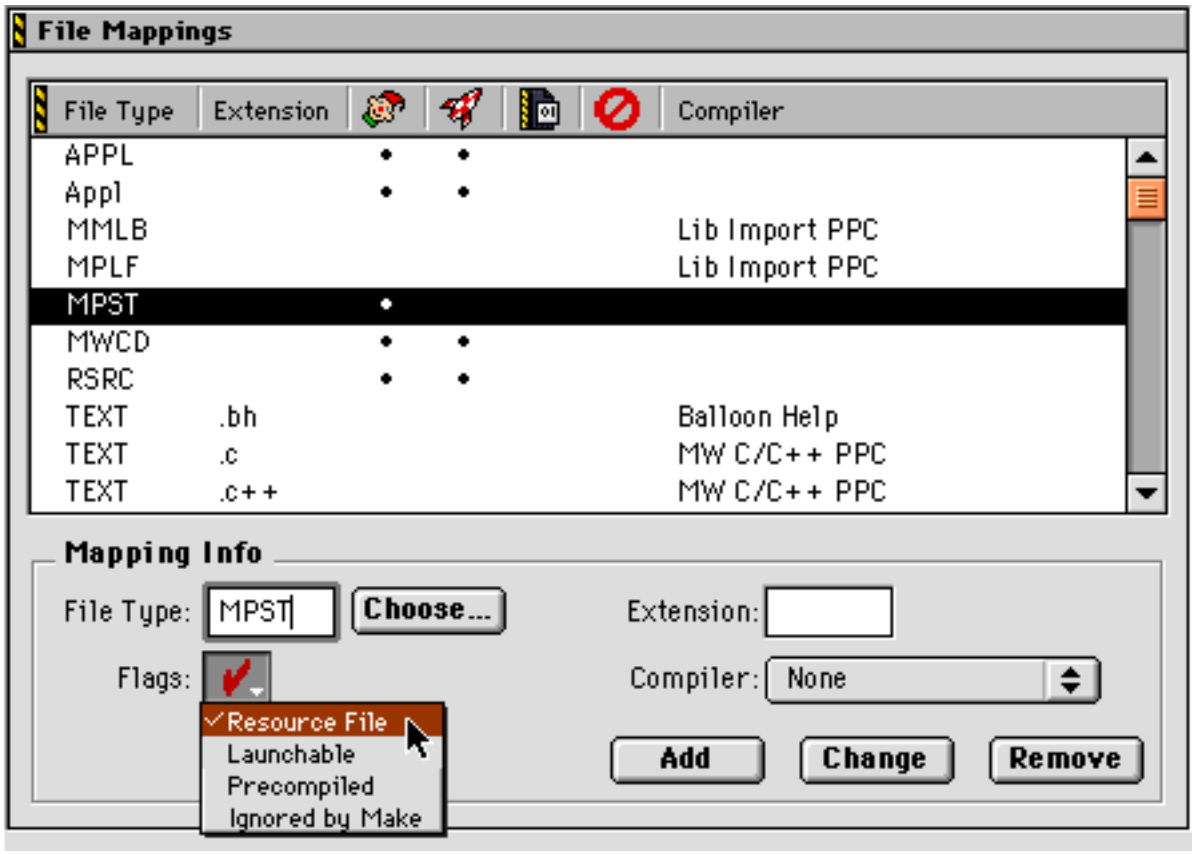
   Add a file type entry with the following properties:

   - Set File Type to ‘MPST’.
   - Set Compiler to **None**.
   - Leave the Extension field blank.

- In the Flags menu, turn on Resource File, and turn off Precompiled, Launchable, and Ignored by Make.

The **File Mappings** panel should look like the one below.

Figure 7.12 MPW Tool entry in the File Mappings panel



Choosing Libraries for an MPW Tool Project

This section describes which libraries to include in 68K and PowerPC MPW Tool projects.

Pascal Libraries

If you are using Pascal to create your MPW Tool, make sure to add `Pascal.tool.68K.lib` or `Pascal.tool.PPC.lib` to your

project for run time calls. You would then use any other non-C and non-C++ libraries as specified in the sections titled "For 68K MPW tool projects" on page 148, and "For PowerPC MPW tool projects" on page 149.

For 68K MPW tool projects

Table 7.1 lists the CodeWarrior libraries you need to add to your 68K MPW Tool projects.

Table 7.1 Choosing which libraries to add to an 68K MPW tools project

If the project…	Add this…
Always	`An appropriate library such as MSL C.68K MPW(NL_4i).Lib and MSL MPWRuntime.68K.Lib`
Calls MPW tool-specific functions	`ToolsLibs.o`
Contains any C++ code	`MSL Runtime68K.Lib`
Calls any Macintosh Toolbox routines	`MacOS.lib`
Calls any math routines	`MathLib68K` *`xx`*`.Lib`

If the project...	Add this...
Contains any C++ code	`MSL C++ xx.Lib` libraries, such as `MSL C++.68K (2i).Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Calls ANSI C standard library routines and contains C++	`MSL C xx.Lib` libraries, such as `MSL C.68K (2i).Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Uses the C++ complex number class	`MSL C++ xx.Lib` libraries, such as `MSL C++.68K (2i).Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.

If the library's name contains *xx*, you need to choose a library based on some **Target Settings** panel options. To learn more about configuring **Target Settings** panel options, refer to the *IDE User Guide*. In a nutshell, the settings you need to be concerned with are 4-byte int and 8-byte double, as well as floating point support and MPW Newlines. You need to make sure that the options you choose in the options panel match those of the library that you're using. For example, you can tell from the library name that

```
MSL C.68K MPW(NL_4i).Lib
```

needs to have 4-byte integers turned on in the options panel, and MPW Newlines turned on.

To learn more information about the Metrowerks library naming conventions, refer to "Library Naming Conventions" on page 421.

For PowerPC MPW tool projects

Table 7.2 lists all the CodeWarrior libraries you might need to add.

Table 7.2 Choosing libraries to add to a PowerPC MPW tool project

If the project…	Add this…
Always	`InterfaceLib`, `MSL MPWCRuntime.Lib` and `MSL C.PPC MPW(NL).Lib`
Calls any MPW tool-specific functions	`PPCToolLibs.o`
Calls any math routines	`MathLib`
Calls any PL string functions	`PLStringFuncsPPC.lib`
Contains any C code and calls any ANSI C standard library routines	`MSL C` *xx*`.Lib` libraries, such as `MSL C.PPC.Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.
Contains any C++ code and calls any ANSI C standard library routines, or uses the C++ complex number class	`MSL C++` *xx*`.Lib` libraries, such as `MSL C++.PPC.Lib`. Refer to "Library Naming Conventions" on page 421 to learn how to choose your library.

Writing an MPW Tool

Listing 7.1 is a `main()` function for a simple MPW tool that prints its arguments. MPW Shell command line parameters and environment variables are provided automatically by the CodeWarrior's MPW runtime library.

Listing 7.1 main() function for an MPW tool

```
#include <stdlib.h>
#include <string.h>
```

```
#include <stdio.h>

void main(int argc, char *argv[], char *envp[])
{
  int i;

  printf("Command line parameters:\n");

  for (i = 0; i < argc; i++)
    printf("%s\n", argv[i]);

  printf("Exported shell variables:\n");

  for (i = 0; envp[i] != NULL; i++)
    printf("%s = %s\n", envp[i],
           envp[i] + strlen(envp[i]) + 1);

  exit(0);
}
```

`main()` takes three arguments as shown in Table 2.26.

Table 7.3 main() arguments for an MPW tool

This arg…	Is…
`argc`	An integer. It is the number of command-line parameters passed to your tool. It's always at least 1 since the first argument is always the tool's name.

This arg…	Is…
`argv`	A pointer to an array of strings. Each string is a command-line parameter. The first element of `argv` the tool's name. The last element is `NULL`.
`envp`	A pointer to an array of the MPW Shell exported variables. Each item in the array is a pointer to an object of this format: *variable-name*\0*variable-value*\0 *Variable-value* immediately follows *variable-name*. *Variable-name* is a C string containing the variable's name. *Variable-value* is a C string containing the variable's value. The last element of `envp` is `NULL`.

Running the MPW Tool

This section explains how to execute your newly created MPW Tool.

1. **Place your compiled tool inside the Tools folder inside the MPW or ToolServer folder on your Mac OS computer.**

   MPW Shell and ToolServer can't use the tool unless it's in the Tools folder.

2. **In MPW or ToolServer, execute the flush command.**

   MPW Shell and ToolServer keep a cache of recently executed tools. The flush command empties that cache, so MPW and ToolServer won't execute an old version of your new tool. You should use this command whenever you are testing a new version of your tool.

3. **Execute your tool.**

   Type the name of the tool in the WorkSheet and press Enter or Command-Return. Here are the lines you might type to execute these operations:

```
flush
WayHappeninMPWToolFAT
```

8

Adding Project Resources

There are three ways CodeWarrior uses resources. This chapter discusses:

- Adding Resource Files to a Project
- Adding Rez files to a Project
- Placing a Resource File in the Project Folder
- Creating a Rez file from a Resource File

You can add other types of resource files with the **File Mappings** panel, described in the *IDE User Guide*.

Adding Resource Files to a Project

CodeWarrior lets you add resource files directly to your project file. When you build the project, CodeWarrior copies the resource files directly into the finished program at link time. By default, a resource file must have the Mac OS Finder file type `'RSRC'` or `'rsrc'`. The following programs create files of these types:

- ResEdit
- Resorcerer
- PowerPlant Constructor

You can also create files of this type using the MPW Tool named Rez. For more information about MPW Tools, refer to the Apple MPW documentation on the CodeWarrior CD.

The CodeWarrior IDE also allows you to build fat applications. For more information on this topic, refer to "Creating Fat Applications" on page 39. and "Creating a Fat MPW tool" on page 140.

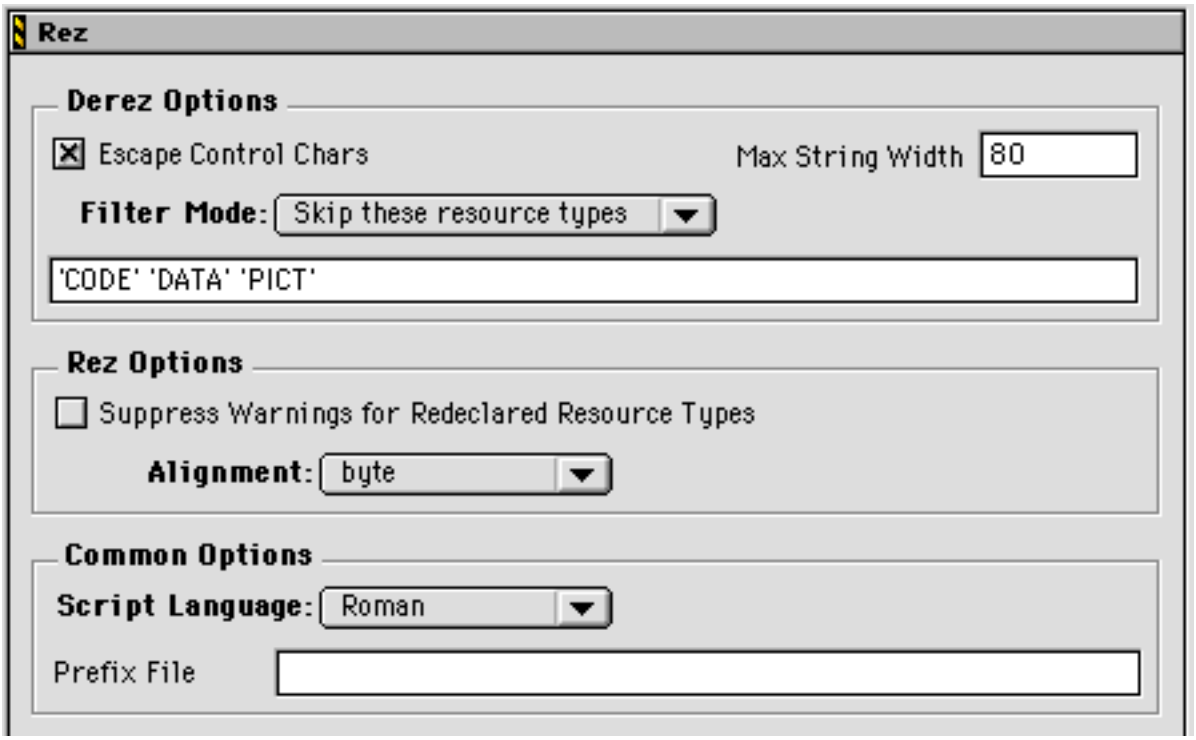
Adding Rez files to a Project

CodeWarrior includes a Rez compiler and lets you add Rez files to your project. By default, a Rez file must have the file type TEXT and its name must end in a `.r extension`.

Rez is a resource description language that lets you describe resources in a text format. The manual *Rez and DeRez* in the CodeWarrior Documentation folder on the CodeWarrior CD describes the syntax.

NOTE: The Rez compiler adds your compiled resources to the project build in the same way that other compilers add compiled source code. It cannot write your resources to an external file. If you need to write resources to a file, the only way to do this is to use ToolServer or MPW. To learn more about differences between the Rez plugin in the CodeWarrior IDE and MPW Rez, refer to "Differences between the Rez Plugin and MPW Rez" on page 158.

Figure 8.1 Rez panel



The **Rez** panel is shown in Figure 8.1. To see this panel, choose the **Target Settings** menu command from the Edit Menu. If the option Suppress Warnings of Redeclared Resource Types is on, the Rez compiler doesn't warn you when you declare the same type more than once. The Prefix File entry field specifies a file that the compiler reads and compiles before each Rez file in the project. It's as if the file were included at the beginning of every Rez file that is processed. Use it for common defines and include files. The alignment option allows you to align the resources to word or longword boundries. This allows the Mac OS Resource Manager to load them faster. The Script Language menu is used for multi-byte character systems.

The Rez compiler doesn't know anything about settings in any other panels. If you need to know something like the processor or project type, define appropriate macros in your prefix file.

Differences between the Rez Plugin and MPW Rez

There are few differences between the Rez Plugin and MPW Rez. Most of the differences help the Rez Plugin work in an integrated environment instead of a command-line-oriented shell. It supports:

- The following macros: `mw_rez` is `1`, `rez` is `1`, and `derez` is `0`.
- Compiling DOS and Windows files. The compiler ignores the line feeds.
- These versions of #pragma once.

  ```
  #pragma once
  #pragma once on
  #pragma once off
  ```

  It ignores all other #pragma statements. For more information on `#pragma` once, refer to the *MW C, C++, and Assembly Language Reference*.
- Including files with `#include <...>`. The Rez Plugin searches the folders in the System pane of the **Access Paths** panel. For more information on this topic, refer to the *IDE User Guide*.
- Using macros in the filename of an `#include` directive. For example:

  ```
  // in prefix file
  #define User_Template_File "MyTemplateFile.r"
  // in .r file
  #include User_Template_File
  ```

If you plan to use the same source file with MPW Rez and CodeWarrior Rez, make sure the `mw_rez` macro is defined before using any of the extensions above.

Placing a Resource File in the Project Folder

You can also include resources in a program by placing a resource file (or an alias to a resource file) in the same folder as your project file. The file must have the same name as your project with .rsrc appended, and the project must not contain any resource files or Rez

files. For example, the resource file for SillyBalls.μ is Silly-balls.μ.rsrc.

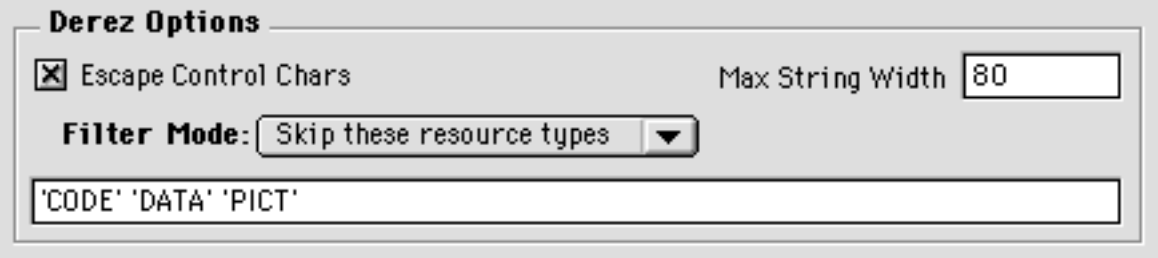
NOTE: The name of the project must contain 25 letters or fewer, so there's room to add the `.rsrc` extension.

WARNING! While this method still works, it really isn't recommended for new projects. It's a much better practice to add resource files explicitly to your project using the Add Files command of the IDE.

Creating a Rez file from a Resource File

The **Rez** panel, as shown in Figure 8.2, contains a section for DeRez options. DeRez allows you to make a Rez source file (of type 'TEXT') from a resource file (`.rsrc`).

Figure 8.2 DeRez options

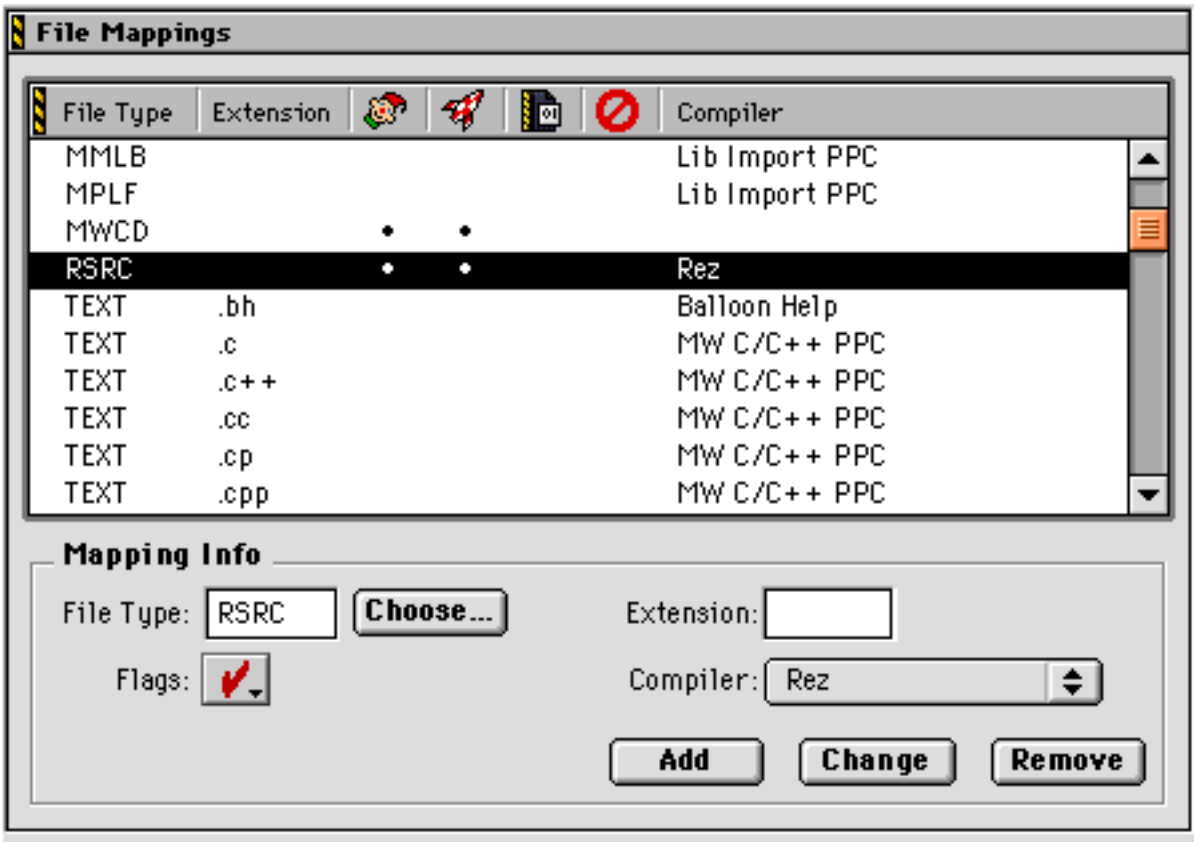


The Filter Mode has two settings: Skip these Resource Types or Only Derez these Resource Types. You can use this to only decompile (derez) specific resources.

Using DeRez

To derez a resource file, change the compiler setting in the **File Mappings** panel for a RSRC file to use the Rez compiler as shown in Figure 8.3 and click the **Change** button.

Figure 8.3 Changing compiler settings



Now click on the resource file you want to decompile in the Project window and choose **Disassemble** from the Project menu. CodeWarrior will create a new window that has a `.dump` extension. You can then save this text file or print out the resource descriptions for publications or other requirements.

NOTE: The resource file type must be 'RSRC' not '`rsrc`' for DeRez to work.

9

Mac OS Merge Settings

This chapter lists the target settings panels for the Mac OS Merge linker configuration when merging 68K and PowerPC applications when creating Fat applications. This chapter includes specific discussion on the **Mac OS Merge** settings panel.

The sections in this chapter are:

- How to Get to Mac OS Merge
- Mac OS Merge Settings Panels
- Mac OS Merge Target Settings Panel

How to Get to Mac OS Merge

You get to the Mac OS Merge configuration and settings panel, the same way you select other PowerPC settings panels: through the PowerPC Target Settings dialog box. To do this:

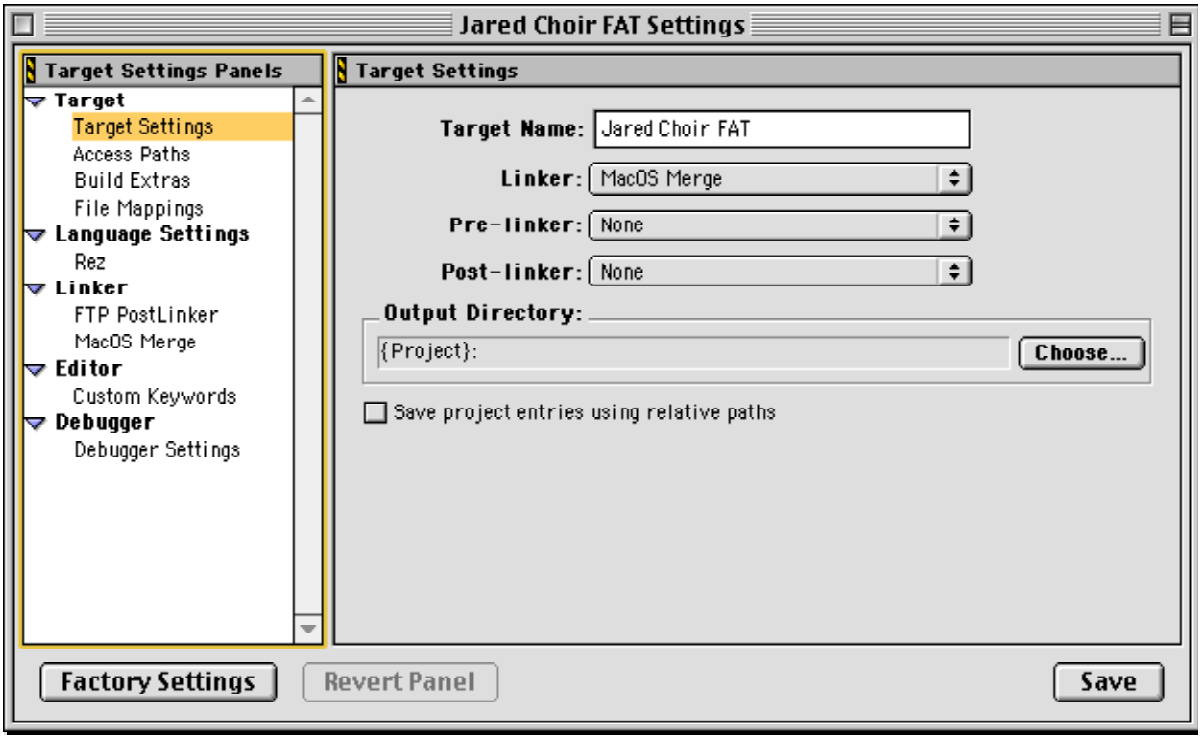
1. **Open a PowerPC project.**

   Open an existing project, or to create a new project, refer to "Creating a New Application Project" on page 35.

2. **Select *Target* Settings from the Edit menu in the IDE.**

   A **Target Settings** dialog box, similar to Figure 9.1, will appear. For information about the **PowerPC Target Settings** dialog box and PowerPC target settings in general, refer to "Target Settings for PowerPC" on page 169.

Figure 9.1 PowerPC Target Settings dialog box

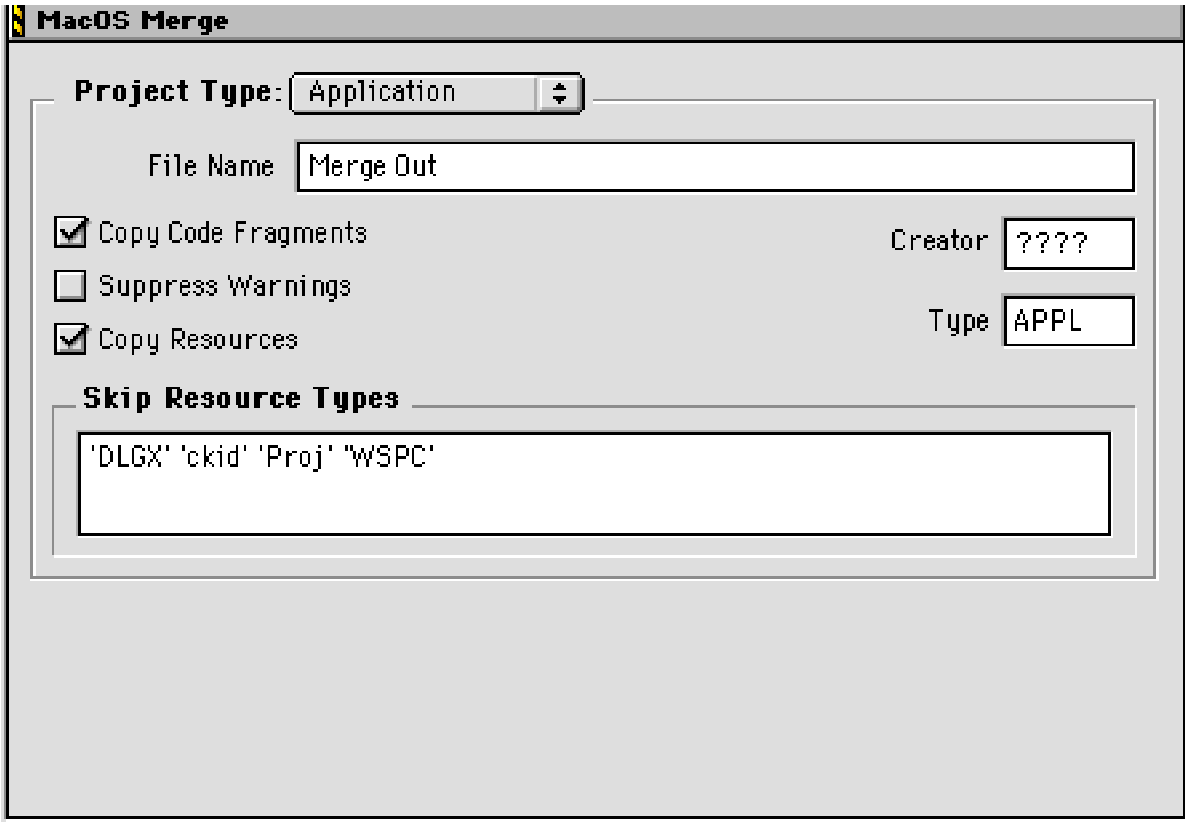


3. **Select the Target Settings panel from the list of panels on the left side of the Target Settings dialog box.**
4. **Choose Mac OS Merge from the Linker pop-up menu on the Target Settings panel, as shown in Figure 9.1.**

   Notice the list of panels on the left side changes. A new panel, **Mac OS Merge**, appears.
5. **Select the Mac OS Merge panel.**

   The panel appears, as in Figure 9.2.

Figure 9.2 The Mac OS Merge target settings panel.



Mac OS Merge Settings Panels

This section lists the Mac OS settings panels and where you can get more information about the different fields and options for these panels.

Table 9.1 Where to get info on Mac OS Merge Target Settings panels

Mac OS Merge panel	Where to get information
Target Settings	Target Settings for PowerPC
Access Paths	*IDE User Guide*

Mac OS Merge panel	Where to get information
Build Extras	*IDE User Guide*
File Mappings	*IDE User Guide*
Rez	Target Settings for PowerPC
Mac OS Merge	This chapter
Custom Keywords	*IDE User Guide*
Debugger Settings	Target Settings for PowerPC

Mac OS Merge Target Settings Panel

The **Mac OS Merge** target settings panel, shown in Figure 9.2, has the following options:

Project Type	File Name
Copy Code Fragments	Suppress Warnings
Copy Resources	Creator
Type	Skip Resource Types

Project Type

Determines what kind of PowerPC program to create. The choices in this pop-up menu are:

- **Application** — creates a Mac OS FAT application
- **Shared Library** — creates a PowerPC FAT shared library
- **Resource File** — creates a PowerPC FAT resource file

File Name

Contains the name of the executable file generated by the IDE.

Copy Code Fragments

Enable this option to copy CFM68K code fragments from the source files into the merged file. When disabled, copying of CFM68K code fragments does not occur.

Suppress Warnings

When enabled, **Suppress Warnings** ignores all duplicate resource types found when merging the source files with the destination file. If disabled, it issues a warning for every duplicate resource it finds during the merge operation.

Copy Resources

Enable Copy Resources to ensure all resources in a 68K file are copied to the destination file. When disabled, no resources will be copied.

Creator

The Finder identifies your application with its **Creator** code, a unique four-character sequence. Enter the code without single quotes. For example, type `SSrt` instead of `'SSrt'`.

Your application's **Creator** code must not conflict with any other application's.

NOTE: To be absolutely sure your Creator code is unique, register it with Apple Computer, Inc., at Macintosh Developer Technical Support. You should be able to find out more information about this topic on the World-Wide Web at `http://devworld.apple.com`

Type

A Mac OS application's **Type** code must always be 'APPL', or 'appe'. The default value for this field is 'APPL'. If you use 'appe', you are writing a background application. Since there are some restrictions for what background applications can do, you should refer to other

sources of information for applications of type 'appe'. One such source would be Apple Technical Notes, accessible on the World-Wide Web at `http://devworld.apple.com`

Other types of projects have different **Type** codes. For example, a code resource might have the Type '`INIT`' or '`PRER`', and a shared library might have the type '`shlb`'.

Enter the code without single quotes. For example, type `APPL` instead of `'APPL'`.

Skip Resource Types

A list of resource types separated by spaces that should be skipped when copying resources from one file to another during a merge operation. Enter your own types as required to customize which resources should not be copied.

10

Target Settings for PowerPC

This chapter discusses each of the settings panels that affect code generation for PowerPC development. By modifying the settings for the individual items within a panel you control the compiler, linker, and other aspects of code generation.

Specific details about how the compiler and linker work for PowerPC development, such as compiler pragmas, linker symbols and so forth, is found in "C and C++ for PowerPC" on page 207 and "Pascal for PowerPC" on page 217.

The sections in this chapter are:

- Target Settings Overview
- Settings Panels for PowerPC

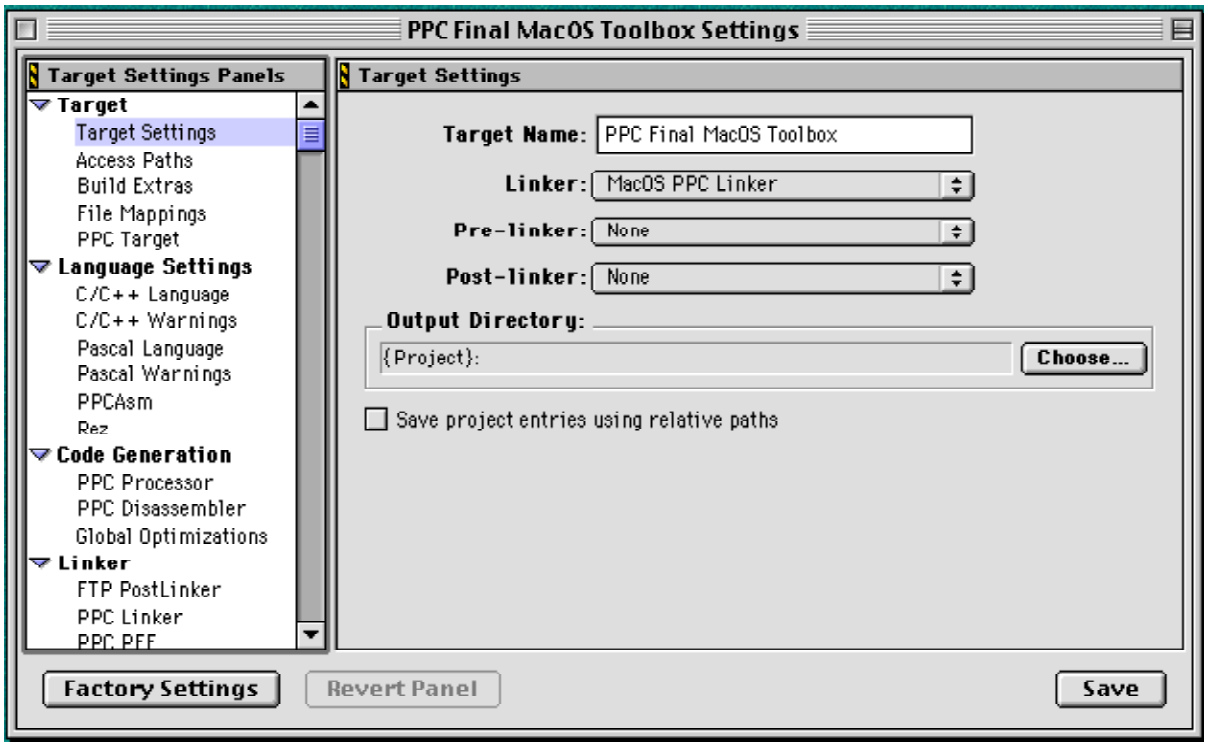
Target Settings Overview

Each target in a CodeWarrior project has its own individual settings. These settings control a variety of features such as compiler options, linker output, error and warning messages, and so forth. You modify these settings through the Target Settings dialog box. This interface is fully explained in the *IDE User Guide*.

In brief, you control compiler and linker behavior for a particular target by modifying settings in the appropriate settings panels in the Target Settings dialog box. To open any settings panel, choose ***Target*** **Settings** from the **Edit** menu, where ***Target*** is the current target in the CodeWarrior project. Or, go to the Target view of the Project window and double-click the target of interest.

When you do, the Target Settings dialog box appears, as shown in Figure 10.1.

Figure 10.1 Target Settings dialog box



Select the panel you wish to see from the hierarchical list of panels on the left side of the dialog box. When you do, that panel appears. You can then modify the settings to suit your needs.

When you modify the settings on a panel, you can restore the previous values by using the **Revert Panel** button at the bottom of the dialog box. To restore the settings to the factory defaults, use the **Factory Settings** button at the bottom of the panel.

TIP: Use project stationery when you create a new project. The stationery has all settings in all panels set to reasonable or default values. You can create your own stationery file with your preferred

settings. Modify a new project to suit your needs, then save it in the stationery folder. See the *IDE User Guide* for details.

Settings Panels for PowerPC

This section discusses those panels that are specific to PowerPC development, and the purpose and effect of each setting. The panels are:

- Target Settings
- PPC Target
- PPC Asm
- Rez
- PPC Processor
- PPC Disassembler
- PPC Linker
- PPC PEF
- Debugger Settings

Settings panels of more general interest are discussed in other CodeWarrior manuals. Table 10.1 lists several panels and where you can find information about them.

Table 10.1 Where to find information on other settings panels

Panel	Manual
Access Paths	*IDE User Guide*
Build Extras	*IDE User Guide*
File Mappings	*IDE User Guide*
C/C++ Language	*C Compilers Reference*
C/C++ Warnings	*C Compilers Reference*
Pascal Language	*Pascal Compilers Reference*

Panel	Manual
Pascal Warnings	*Pascal Compilers Reference*
Global Optimizations	*IDE User Guide*
Custom Keywords	*IDE User Guide*

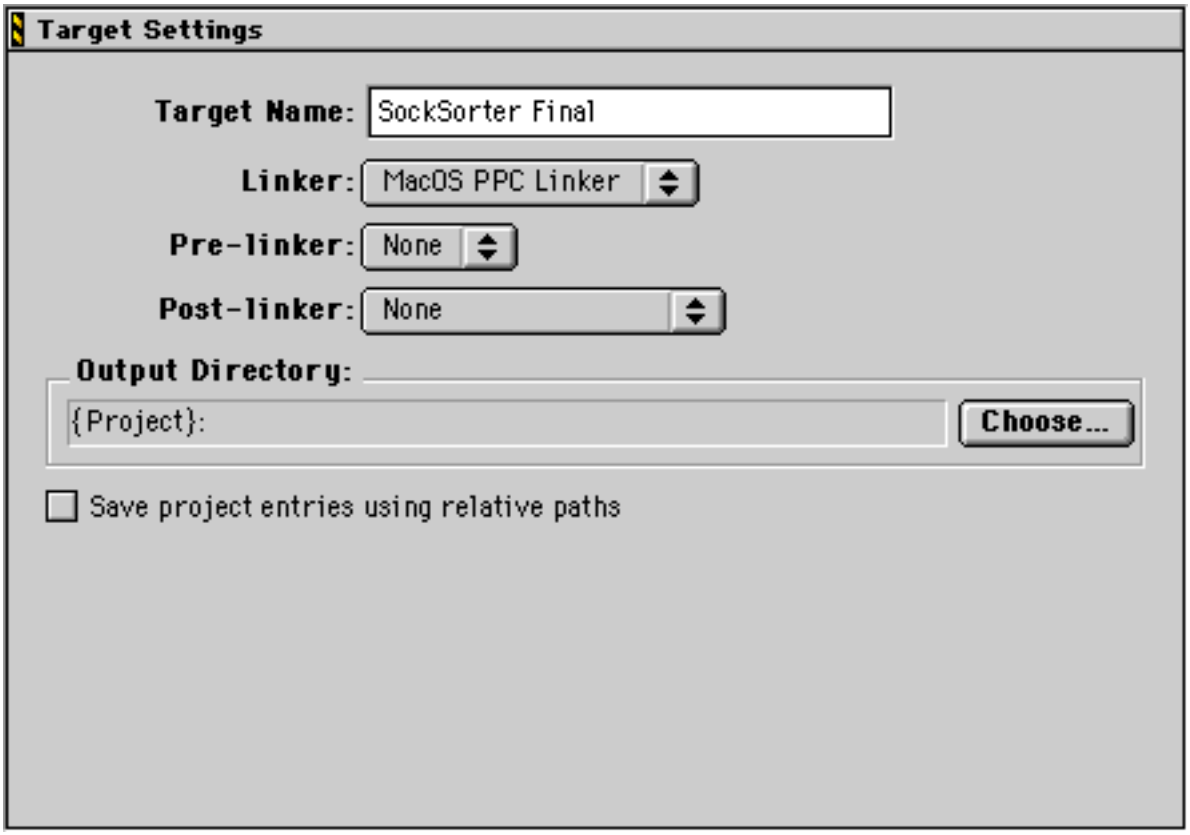
Target Settings

The Target Settings *dialog box* contains a Target Settings *panel*. The dialog box and the panel are not the same. The dialog box displays all panels, one at a time. The Target Settings *panel* is one of those panels.

The Target Settings panel, shown in Figure 10.2, is perhaps the most important panel in CodeWarrior. This is the panel where you pick your target. When you select a linker in the Target Settings panel, you specify the target operating system and/or chip. The other panels listed in the Settings dialog box will change to reflect your choice.

Because the linker choice affects the visibility of other related panels, you must set your target first before you can specify other target-specific options like compiler and linker settings.

Figure 10.2 The Target Settings panel



NOTE: The Target Settings panel is not the same as the PPC Target panel. You specify the target in the Target Settings panel. You set other target-specific options in the PPC Target panel.

The items in this panel are:

- Target Name
- Linker
- Pre-Linker
- Post-Linker
- Output Directory
- Save Project Entries Using Relative Paths

Target Name

Use the **Target Name** text field to set or change the name of a target. When you use the Targets view in the Project window, you will see the name that you have set.

The name you set here is *not* the name of your final output file. It is the name you assign to the target for your personal use. The name of the final output file is set in the PPC Target panel.

Linker

Choose a linker from the items listed in the **Linker** pop-up menu. For PowerPC, use **Mac OS PPC Linker**.

You may also choose the **MacOS Merge** option from the menu if you wish to create a Fat application. See for information about the **MacOS Merge** panel. See "Creating Fat Applications" on page 39 for information about creating Fat applications.

Pre-Linker

Some targets have pre-linkers that perform work on object code before it is linked. There is no pre-linker for PowerPC development.

Post-Linker

Some targets have post-linkers that perform additional work (such as object code format conversion) on the final executable. There is no post-linker for PowerPC development.

Output Directory

This is the directory where your final linked output file will be placed. The default location is the directory that contains your project file. Click the Choose button to specify another directory.

Save Project Entries Using Relative Paths

To add two or more files with the same name to a project, select this option. When this option is off, each project entry must have a unique name.

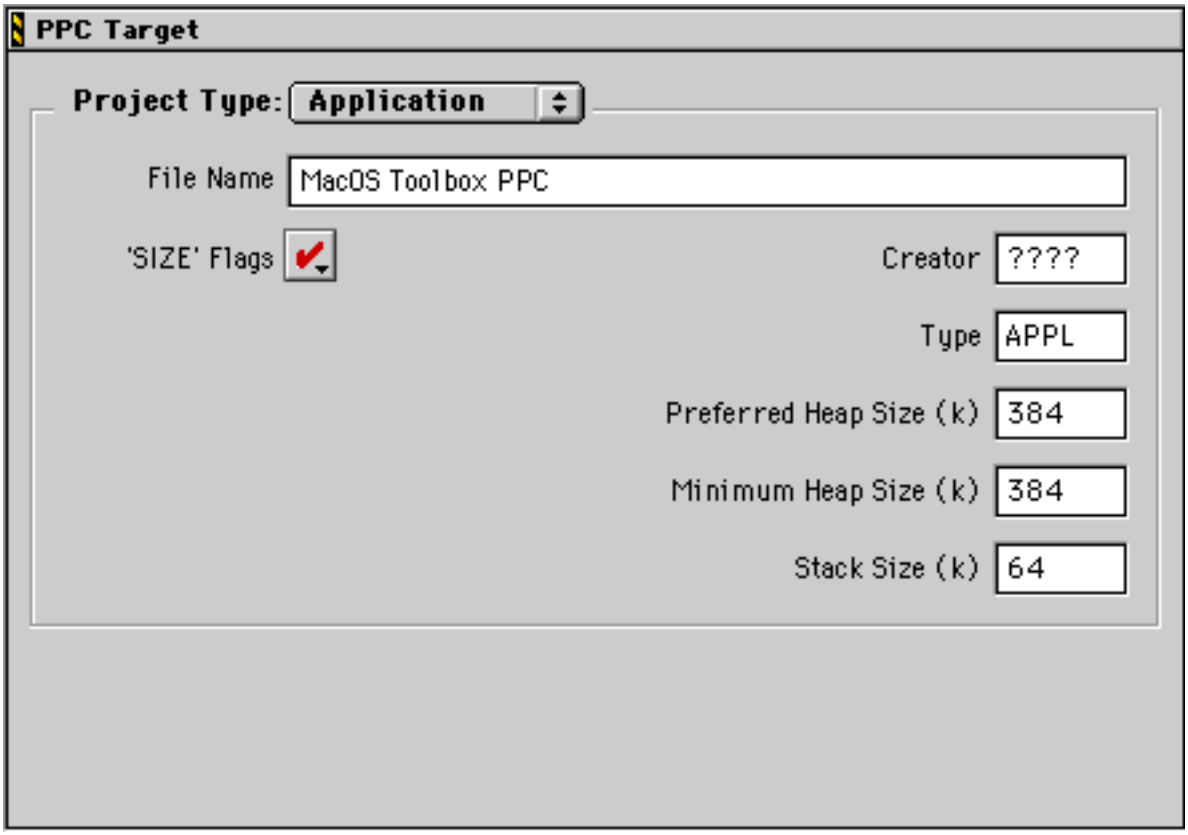
When this option is selected, the IDE includes information about the path used to access the file as well as the file name when it stores information about the file. When searching for a file, the IDE combines **Access Path** settings with the path settings it includes for each project entry.

When this option is off, the IDE only records information about each project entry's file name. When searching for a file, the IDE only uses Access Paths.

PPC Target

The **PPC EABI Project** panel, shown in Figure 10.3, is where you set the name of your final output file.

Figure 10.3 The PPC Target panel.



The items in this panel are:

Project Type
File Name
'SIZE' Flags
Creator
Type
Preferred Heap Size
Minimum Heap Size
Stack Size
Sym Name
Resource Name
ResType
ResID
Display Dialog
Merge to File
Resource Flags
Header Type

Project Type

Determines what kind of PowerPC program to create. The choices in this pop-up menu are:

- **Application**—creates a Mac OS PowerPC application
- **Shared Library**—creates a PowerPC PEF shared library
- **Code Resource**—creates a PowerPC code resource
- **Static Library**—creates a CodeWarrior library to use in other projects

File Name

Contains the name of the application generated by the IDE.

This option appears for every type of PowerPC project selected form the **Project Type** pop-up menu.

'SIZE' Flags

Specifies the contents of the 'SIZE' generated and included in the application. See Table 10.2 for information on the items in this pop-up menu.

This option appears if **Application** is selected from the **Project Type** pop-up menu.

Table 10.2 The 'SIZE' resource options

Turn this option on…	If your application…
`acceptSuspend-ResumeEvents`	Handles suspend and resume events as well as deactivate and activate events. These events are sent when your application is brought to the foreground or sent to the background.
`canBackground`	Supports null events when it is in the background. If you turn this option off, the Finder sends your application only update events.
`doesActiveOnFGSwitch`	Does not need to receive an activate (or deactivate) event after it has come to the foreground (or background).
`onlyBackground`	Contains no user interface and is meant to run in the background.
`getFrontClicks`	Supports front clicks which bring it to the foreground. Front clicks are mouse-down and mouse-up events.
`acceptAppDiedEvents`	Needs to know whether the application which it launched was quit or crashed.
`is32BitCompatible`	Is 32-bit clean.
`isHighLevelEventAware`	Supports AppleEvents or other high level events. If you turn this option off, the Finder does not send high level events to your application.

Turn this option on…	If your application…
`localAndRemoteHLEvents`	Handles Apple Events and other high-level events from remote computers. If you turn this option off, the Finder does not send your application AppleEvents from remote computers.
`isStationeryAware`	Supports stationery documents. If you turn this option off, the Finder handles stationery documents for you.
`useTextEditServices`	Supports inline text services introduced with System 7.
`isDisplayManagerAware`	Handles the Display Notice event, which tells your application to move its windows after the monitor settings have changed. If you turn this option off, the Display Manager moves your application's windows so they don't disappear off the screen.

Creator

The Finder identifies your application with its **Creator** code, a unique four-character sequence. Enter the code without single quotes. For example, type `SSrt` instead of `'SSrt'`.

Your application's **Creator** code must not conflict with any other application's.

NOTE: To be absolutely sure your Creator code is unique, register it with Apple Computer, Inc., at Macintosh Developer Technical Support. You should be able to find out more information about this topic on the World-Wide Web at `http://devworld.apple.com`

This option appears if **Application**, **Shared Library**, or **Code Resource** is selected from the **Project Type** pop-up menu.

Type

A Mac OS application's **Type** code must always be 'APPL', or 'appe'. The default value for this field is 'APPL'. If you use 'appe', you are writing a background application. Since there are some restrictions for what background applications can do, you should refer to other sources of information for applications of type 'appe'. One such source would be Apple Technical Notes, accessible on the World-Wide Web at `http://devworld.apple.com`

Other types of projects have different **Type** codes. For example, a code resource might have the Type 'INIT' or 'PRER', and a shared library might have the type 'shlb'.

Enter the code without single quotes. For example, type APPL instead of `'APPL'`.

This option appears if **Application**, **Shared Library**, or **Code Resource** is selected from the **Project Type** pop-up menu.

Preferred Heap Size

The recommended amount of RAM your Mac OS application wants, in KBytes. In the Finder's Get Info dialog, this is the number in the Suggested Size field and the default value for the Preferred Size field.

This option appears if **Application** is selected from the **Project Type** pop-up menu.

Minimum Heap Size

The minimum amount of RAM your application needs, in KBytes. In the Finder's Get Info dialog, this is the default value for the Minimum Size field.

This option appears if **Application** is selected from the **Project Type** pop-up menu.

Stack Size

An application's stack contains local variables, return addresses, and temporary information. If your application uses a lot of stack space (for example, it makes extensive use of recursive functions, or uses large arrays local to a routine), you may need to change the default size of the stack.

In a PowerPC application, you can set the size in the **Stack Size** field. The default size is 64K.

In a PowerPC application, you must set the stack size within your application's code by calling the function `SetApplLimit()`. For more information, see Chapter 1, "Introduction to Memory Management," in *Inside Macintosh: Memory*.

This option appears if **Application** is selected from the **Project Type** pop-up menu.

Sym Name

Specifies the name of the file in which the code resource's SYM information (debugging information) will be placed.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

Resource Name

Specifies the name of the code resource's name in the resource fork.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

NOTE: If you're creating a multi-segment resource, the linker only names the main resource, not its segments.

ResType

Identifies what kind of resource you're creating. For example, `'XCMD'` is for a HyperCard XCMD, `'XDEF'` is for a window definition, and `'MDEF'` is for a menu definition. Enter the code without single quotes. For example, type `XDEF` instead of `'XDEF'`.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

ResID

The ID number, along with the resource type, uniquely identifies the resource to the system software. Consult *Inside Macintosh:Toolbox Essentials* for more information on choosing a resource ID.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

Display Dialog

Shows a dialog box prompting you for a file to store the code resource in. If this item is selected, the file entered in File Name is ignored. If **Merge To File** is selected, the code resource will be added to file you've selected. If **Merge To File** is deselected, the IDE will delete the contents the file (or creates the file if it doesn't already exist) you've selected before adding the code resource.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

Merge to File

Specifies that the code resource should be added to the file specified by File Name or selected in the file dialog box when Display Dialog is selected. If selected, the code resource is added to the file. If deselected, the IDE deletes the file's contents (or creates the file if it doesn't already exist) before adding the code resource.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

Resource Flags

Specifies the Resource Manager flags to set for the resource. Table 10.3 describes what each resource attribute flag does. For more information on these flags, refer to the chapter called "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox,* published by Addison-Wesley.

Table 10.3 Resource attributes

Turn this attribute on…	To…
System Heap	Place the resource in the system heap. If your resource is too large, it is placed in the application heap. If you turn this option off, your resource is placed in the application heap.
Purgeable	Purge the resource from memory when the system needs to make more memory available.
Locked	Lock the resource in the heap so the system cannot move it. It gets locked into the heap's low memory address space.
Protected	Protect the contents, name, and ID of the resource. If a resource is protected, it cannot be removed from the file to which it belongs.
Preload	Load the resource into memory as soon as the Resource Manager opens it. When the application is launched, anything set with preloaded is loaded.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

Header Type

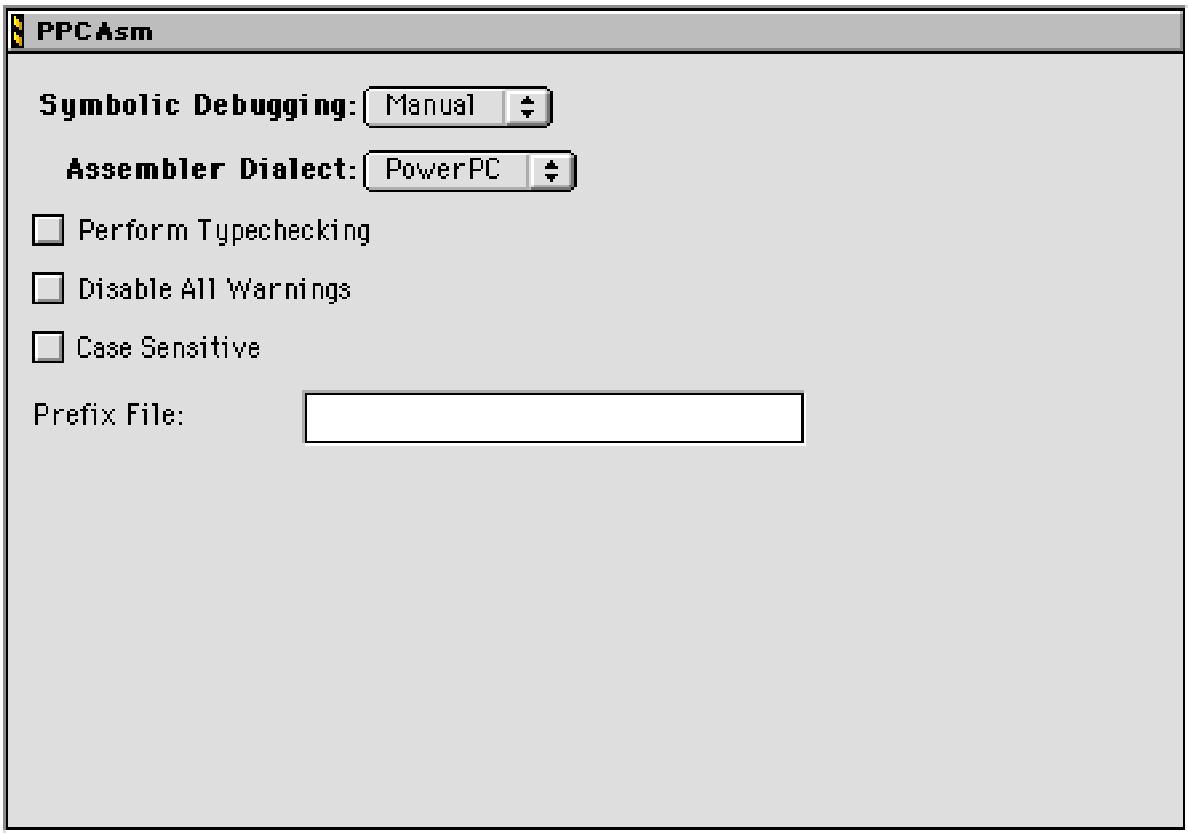
Specifies the type of code resource header to place at the beginning of the code resource.

This option appears if **Code Resource** is selected from the **Project Type** pop-up menu.

PPC Asm

The **PPC Asm** settings panel, shown in Figure 10.4, allows you to configure the language that the assembler compiles.

Figure 10.4 The PPC Asm panel.



The settings are:

Symbolic Debugging	Assembler Dialect
Perform Typechecking	Disable All Warnings
Case Sensitive	Prefix File

Symbolic Debugging

The **Symbolic Debugging** pop-up menu allows you to choose how the assembler debug information is generated. There are two options in this menu:

- **Auto** — the assembler will auto-generate debug information
- **Manual**— allows you to code debug information yourself

Assembler Dialect

The **Assembler Dialect** pop-up menu allows you to choose the processor architecture for which your assembly language is targeted. There are three options listed in the menu. Choose **PowerPC** for Mac OS code generation.

WARNING! If you choose an option other than **PowerPC** from the **Assembler Dialect** pop-up menu, your code will not run correctly on the Mac OS system.

Perform Typechecking

When selected, the **Perform Typechecking** option checks operand types for instructions.

Disable All Warnings

When selected, the **Disable All Warnings** checkbox turns off warnings. You will only see errors.

Case Sensitive

When selected, the **Case Sensitive** option determines whether or not the assembler will accept a label with a different case.

Prefix File

The **Prefix File** edit field allows you to specify the file to be read in by the assembler before it processes the source file.

Rez

The **Rez** settings panel, shown in Figure 10.5, allows you to control how rez compiles a source file, and how derez decompiles a resource file.

The settings are:

Escape Control Chars	Max String Width
Filter Mode	Suppress Warnings for Redeclared Resource Types
Alignment	Script Language
Prefix File	

Escape Control Chars

When selected, the **Escape Control Chars** checkbox allows control characters to be printed.

Max String Width

The **Max String Width** edit field allows you to specify the maximum string width of the derez output. You must specify a width between two and 120. If you enter a number outside this range, derez will give you a warning and use the default value of 80.

Filter Mode

The **Filter Mode** edit field allows you to specify the resource types to be derezed or the resource types not to be derezed, depending on which option is selected from the Filter Mode pop-up menu. You may enter any four-character type in this field. Types must be surrounded by single quotes (') and separated by commas (,). There are two options in the pop-up menu:

- **Skip these resource types** — if selected, the types entered in the edit field will not be derezed; every other type will be derezed.
- **Only derez the resource types** — if selected, the types specified in the edit field will be the only types to be derezed; all others will be skipped.

Suppress Warnings for Redeclared Resource Types

When selected, the **Suppress Warnings for Redeclared Resource Types** checkbox will not allow warnings to be shown for redeclared resource types.

Alignment

The **Alignment** pop-up menu is currently not supported.

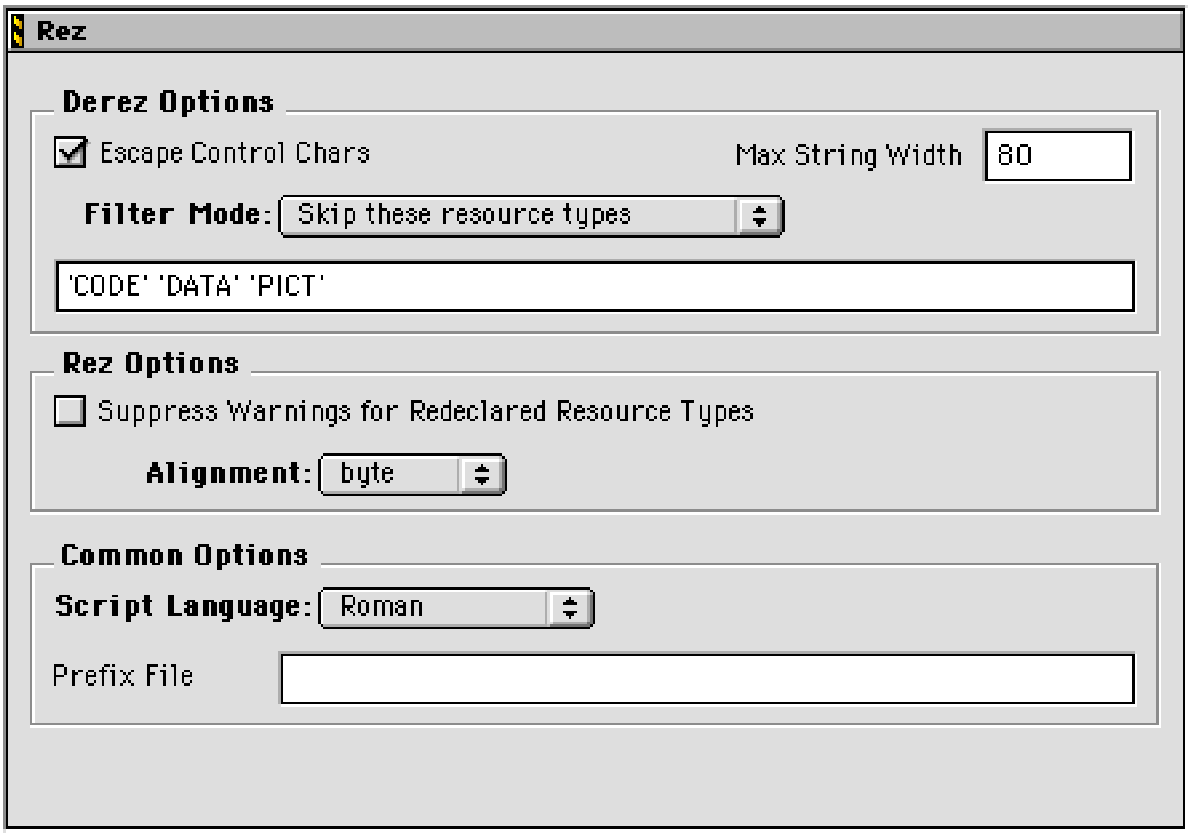
Script Language

The **Script Language** pop-up controls which script system Rez uses when it compiles multi-byte characters. For example, if you have the Japanese Language Kit installed and your strings contain Kanji characters, then you would need to change the script language from **Roman** to **Japanese**.

Prefix File

The **Prefix File** edit field allows you to specify the file to be read in by the compiler before it processes the source file or the resource file in the case of derez. The prefix file is where derez gets the resource type definitions that it uses to decompile the resources.

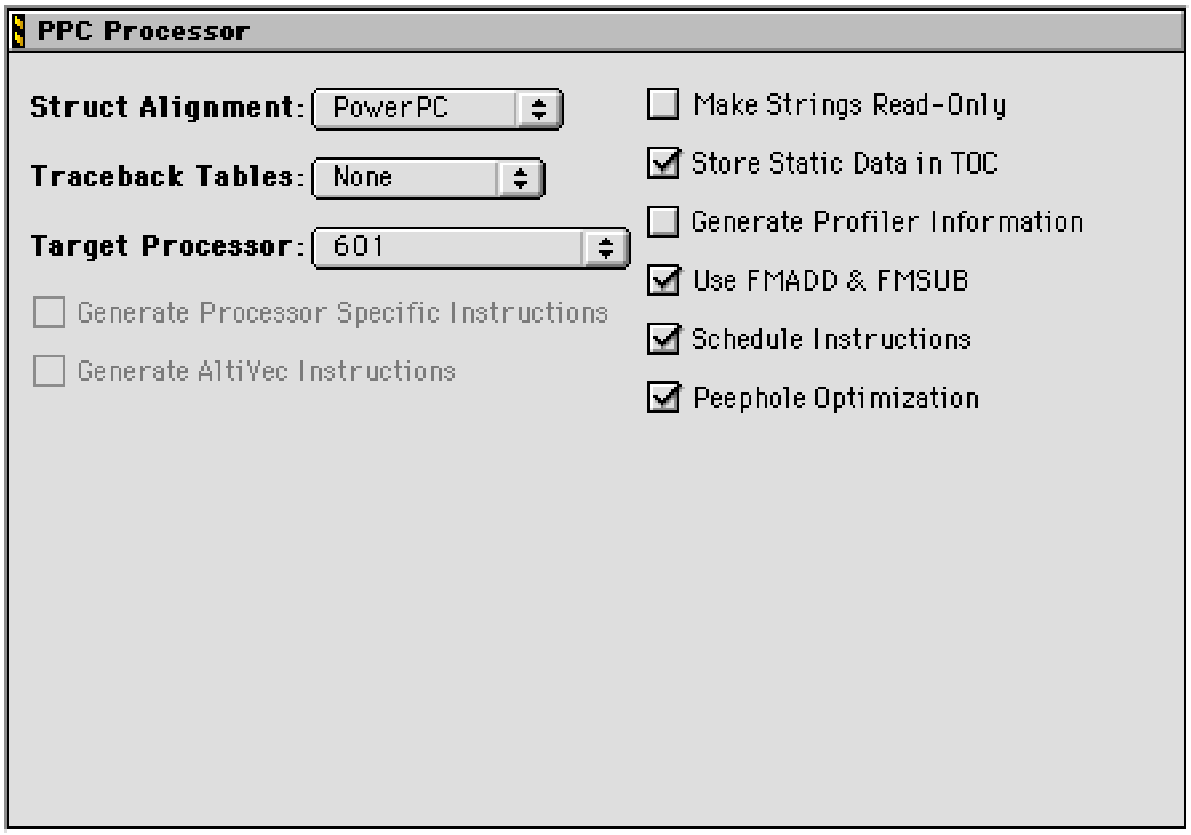
Figure 10.5 The Rez settings panel.



PPC Processor

The **PowerPC Processor** panel, shown in Figure 10.6, is where you control settings related to code generation for the PowerPC platform.

Figure 10.6 The PPC Processor panel.



The items in this panel are:

- Struct Alignment
- Traceback Tables
- Target Processor
- Make Strings ReadOnly
- Store Static Data in TOC
- Generate Profiler Information
- Use FMADD & FMSUB
- Schedule Instructions
- Peephole Optimization

Struct Alignment

Specifies how records and structures are arranged in memory. Table 10.4 describes the options in this pop-up menu.

Table 10.4 Structure alignment options

If you choose…	The linker aligns every field on…
68K	2-byte boundaries, unless a field is only 1-byte long. This is the standard alignment for 68K Macintosh computers.
68K 4-byte	4-byte boundaries.
PowerPC	Its natural boundary. For example, it aligns a 1-byte character on a 1-byte boundary and a 16-bit integer on a 2-byte boundary. The compiler applies this alignment recursively to structured data and arrays containing structured data. So, for example, it aligns an array of structured types containing an 8-byte floating point member on an 8-byte boundary. This is the standard alignment for PowerPC computers.

This option corresponds to `#pragma options align`.

Traceback Tables

Determines if extra data for debugging should be generated. If this option is on, the compiler generates an AIX-format traceback table for each routine, and places these tables in the executable code. Both CodeWarrior and Apple debuggers are able to use these traceback tables. This option corresponds to #pragma traceback. There are three options in this menu:

- **None** — no traceback tables are generated
- **Inline** — traceback tables are generated in the AIX-format, with the traceback tables stored immediately after each function
- **Out of Line** — traceback tables are generated in the MrC format, with a single long value following each function that serves as a pointer to the rest of the traceback data. The vast majority of the traceback data is stored somewhere else.

Target Processor

The **Target Processor** pop-up menu selects which processor you are targeting. Choose **Generic** if the processor you are working with is not listed on this menu, or you want to generate code that runs on any PowerPC processor.

This menu determines if the compiler should rearrange instructions to take advantage of a specific processor's architecture. If a processor is selected in the **Target Processor** pop-up menu, scheduling of instructions is optimized for that specific processor.

Make Strings ReadOnly

Determines where to store character string constants. If this option is off, the compiler stores string constants in the data section (class `RW`). If this option is on, the compiler stores string constants in the code section (class `RO`).

Variables that are not initialized to the address of another object at runtime are always placed in the code section, including C/C++ variables declared with the `const` storage-class modifier.

This option corresponds to `#pragma readonly_strings`.

Store Static Data in TOC

Determines where to store static variables. When this option is on, the compiler makes object code smaller and faster by storing static variables that are 4 bytes or smaller directly in the code fragment's TOC (Table Of Contents). When this option is off, the compiler allocated memory normally for all static data and stores pointers to the data in the TOC.

This option corresponds to `#pragma toc_data`.

Generate Profiler Information

Determines if special object code should be generated to collect information for a code profiler. If this option is on, the compiler generates instructions that collect data during runtime for the CodeWar-

rior Profiler. If this option is off, the compiler doesn't generate extra instructions for the profiler.

This option corresponds to `#pragma profile`.

Use FMADD & FMSUB

Determines whether or not the compiler uses special instructions to speed up floating-point computations. If this option is on, the compiler uses special floating-point instructions, including FMADD, FMSUB, and FNMAD to speed up floating-point computations. Keep in mind that some kinds of computations give unexpected results when this option is on (Listing 10.1). When this option is off, `Y` and `Z` have the same value.

Listing 10.1 Using FMADD and FMSUB instructions

```
register double A, B, C, D, Y, Z;
register double T1, T2;

A = C = 2.0e23;
B = D = 3.0e23;
Y = (A * B) - (C * D);
printf("Y = %f\n", Y);
/* Output: 21267700587560961875633692999968.000000 *)

T1 = (A * B);
T2 = (C * D);
Z = T1 - T2;
printf("Z = %f\n", Z);
/* Output: 0.000000 *)
```

This option corresponds to `#pragma fp_contract`.

Schedule Instructions

Determines if the compiler should rearrange instructions to take advantage of a specific processor's architecture. If a processor is selected in the **Instruction Scheduling** pop-up menu, scheduling of instructions is optimized for that specific processor. If **Off** is se-

lected, the compiler doesn't use the Instruction Scheduling optimization.

NOTE: Enabling instruction scheduling can make source-level debugging more difficult (because the source code may not correspond exactly to the underlying instructions being run.) It is sometimes helpful to turn this option off when debugging, and then turn it on once you have finished the bulk of your debugging.

Peephole Optimization

The **Peephole Optimization** checkbox controls whether the compiler performs "peephole" optimizations, which are small local optimizations that eliminate some compare instructions and improve branch sequences.

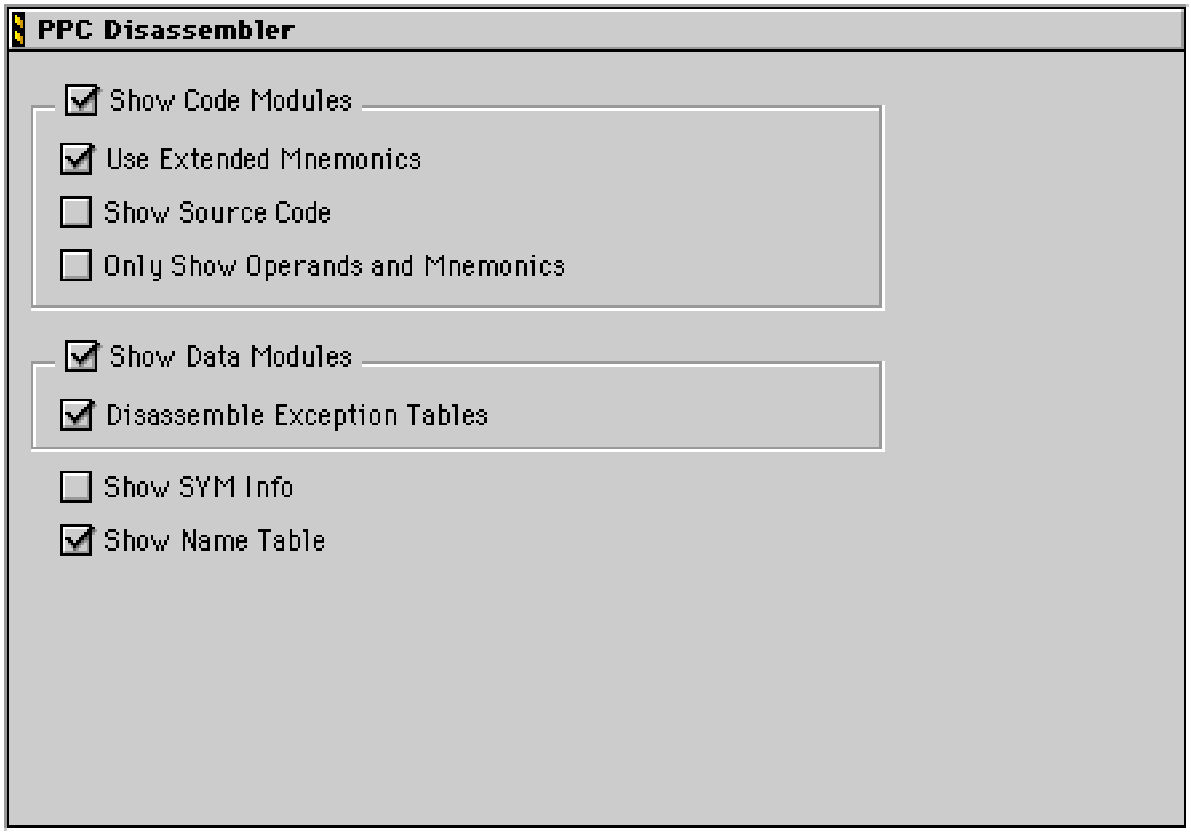
This option corresponds to `#pragma peephole`. By default, this option is off.

PPC Disassembler

The PowerPC Disassembler panel, shown in Figure 10.7, is where you control the information displayed when you choose the **Disassemble** command from the IDE's **Project** menu.

See the "Compiling and Linking" chapter of the *IDE User Guide* for general information about the Disassemble command.

Figure 10.7 The PPC Disassembler panel.



The items in this panel are:

Show Code Modules	Use Extended Mnemonics
Show Source Code	Only Show Operands and Mnemonics
Show Data Modules	Disassemble Exception Tables
Show SYM Info	Show Name Table

Show Code Modules

The **Show Code Modules** checkbox determines whether the disassembler outputs the PEF code sections for the module being disassembled.

If the **Show Code Modules** checkbox is selected, the **Use Extended Mnemonics** and **Only Show Operands and Mnemonics** checkboxes appear.

Use Extended Mnemonics

The **Use Extended Mnemonics** checkbox determines whether the disassembler lists the extended mnemonics for each instruction for the module that was disassembled.

This checkbox is displayed only if the **Show Code Modules** checkbox is selected.

Show Source Code

The **Show Source Code** checkbox determines whether the disassembler lists the source code that corresponds to the disassembled instructions.

This checkbox is displayed only if the **Show Code Modules** checkbox is selected.

Only Show Operands and Mnemonics

The **Only Show Operands and Mnemonics** checkbox determines whether the disassembler lists the offset for any functions in the module that was disassembled.

This checkbox is displayed only if the **Show Code Modules** checkbox is selected.

Show Data Modules

The **Show Data Modules** checkbox determines whether the disassembler outputs any PEF data sections for the module being disassembled.

If the **Show Data Modules** checkbox is selected, the **Disassemble Exception Tables** checkbox appears.

Disassemble Exception Tables

The **Disassemble Exception Tables** checkbox determines whether the disassembler outputs any C++ exception tables for the module that was disassembled.

This checkbox is displayed only if the **Show Data Modules** checkbox is selected.

Show SYM Info

The **Show SYM Info** checkbox tells the disassembler to include symbol information in the disassembled output.

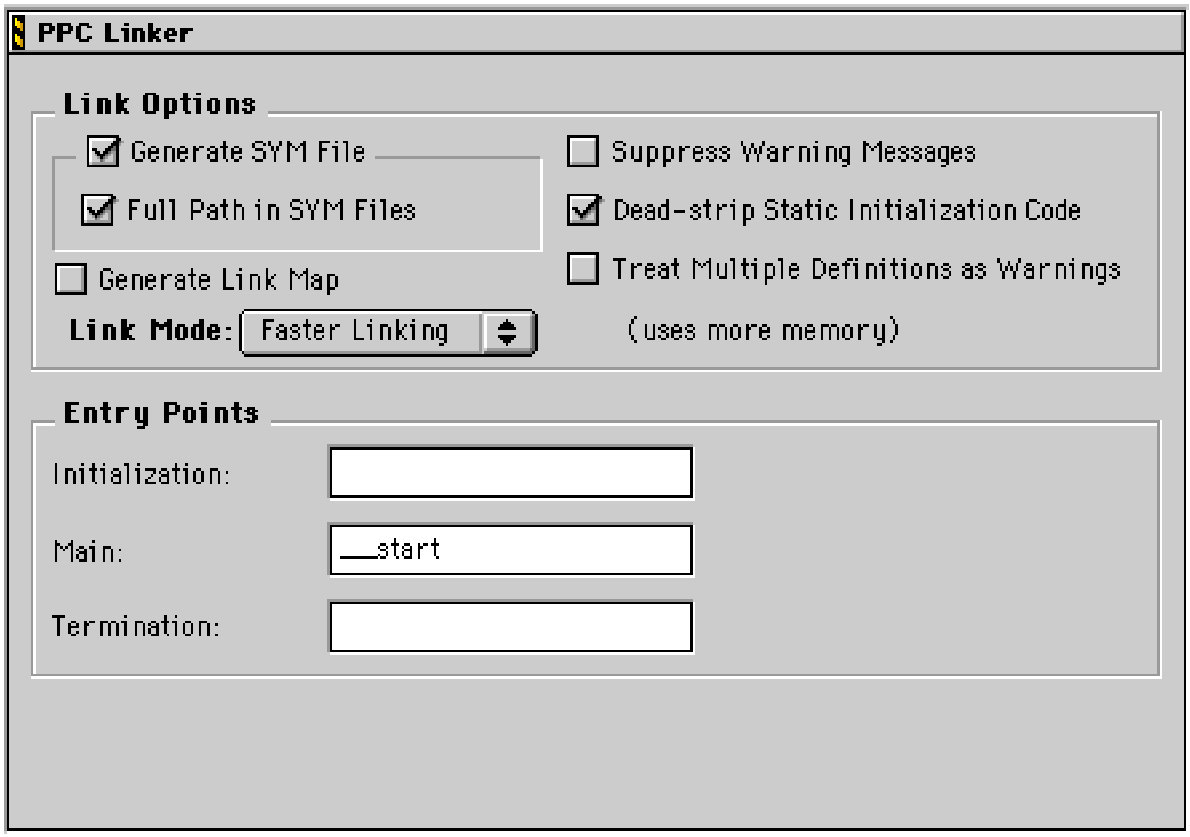
Show Name Table

The Show Name Table checkbox tells the disassembler to list the symbols in the object code.

PPC Linker

The **PPC EABI Linker** panel, shown in Figure 10.8, is where you control settings related to linking your object code into final form, be it executable, library, or other type of code.

Figure 10.8 The PPC Linker panel.



These items in this panel are:

Generate SYM File	Full Path in SYM Files
Generate Link Map	Link Mode
Suppress Warning Messages	Dead-strip Static Initialization Code
Treat Multiple Definitions as Warnings	Initialization
Main	Termination

Generate SYM File

Creates a SYM file for your project the next time it makes your project. To determine which source files CodeWarrior includes in

that SYM file, use the Debug Info markers in the Project window. To learn how to set these markers, refer to the *IDE User Guide*.

Use `#pragma sym on` to generate SYM information for subsequent functions. Use `#pragma sym off` to stop generating SYM information.

Full Path in SYM Files

Puts the full pathname of the source file in the SYM file. Turning on this option is recommended. However, it slightly increases the size of your SYM file. If this option is off, the debugger might prompt you to locate a source file if it can't find it in the same folder as the SYM file that refers to it.

This checkbox is displayed only if the **Generate SYM File** checkbox is selected.

Generate Link Map

The **Generate Link Map** checkbox controls whether the linker generates a link map. When this setting is on the linker generates a link map. When this setting is off, the linker does not generate a link map.

The file name for the link map adds the extension `.MAP` to the linker's generated file name. The file is put in the same folder as the CodeWarrior project file.

The link map shows which file provided the definition for every object and function in the output file. It also displays the address given to each object and function, a memory map of where each section will reside in memory and the value of each linker generated symbol. Although the linker aggressively strips unused code and data when the relocatable file is compiled with the CodeWarrior compiler, it never dead strips assembler relocatables or relocatables built with other compilers. If a relocatable wasn't built with the CodeWarrior C/C++ compiler, the link map lists all of the unused but unstripped symbols. You can use that information to remove the symbols from the source and rebuild the relocatable in order to make your final process image smaller.

Link Mode

Controls how much memory the IDE uses to improve linking performance.

Choosing **Faster Linking** gives the best linking performance, but requires a larger memory partition and more temporary memory. When using this option you may need to increase the IDE's memory partition and possibly quit other applications.

Choosing **Normal Linking** gives moderate performance and uses less memory than the Faster Linking option.

Choosing **Slower Linking** gives the slowest performance but uses the least amount of memory of all linking modes.

Suppress Warning Messages

The **Suppress Warning Messages** checkbox controls whether the linker displays warnings. When this setting is on, the linker displays warnings in the CodeWarrior message window. When this setting is off, the linker does not display warnings.

Dead-strip Static Initialization Code

Removes unreferenced runtime initialization and object constructor instructions.

If the **Dead-strip Static Initialization Code** option is in, the linker removes runtime initializations and constructor calls for all the static and global objects in a file, if your program does not refer to any variables or functions in that file.

For example, suppose that a project contains the files A.cp, B.cp, and C.cp. The `main()` function in A.cp calls functions in B.cp but not in C.cp, and C.cp has some global variables of types which have constructors. The compiler generates code called a static initializer which calls the constructors for the variables in C.cp.

If this option is on, the linker recognizes that no references are made to any of C.cp's variables, so the linker does not link the variables or the static initializers for them into the program.

If this option is off, the linker leaves the variables and the static initializer in, and it may end up linking in variables and functions that the program never uses.

In general, leave this option on to produce smaller and faster code. However, if your program expects the static initializers to be executed for their side-effects, turn this option off.

Treat Multiple Definitions as Warnings

Specifies if the linker should warn about items that have the same name. If this option is on, the linker issues a warning if more than one item in the object code has the same name. If this option is off, the linker issues an error.

Initialization

Specifies the symbol name of the routine to call when the CFM68K fragment is loaded into memory. PowerPC calls this routine before referring to any other items in the container. Typically, the initialization routine prepares and sets up objects and services that the fragment will use when its main routine is called.

Main

Specifies the symbol name of the routine that is the fragment's main entry point.

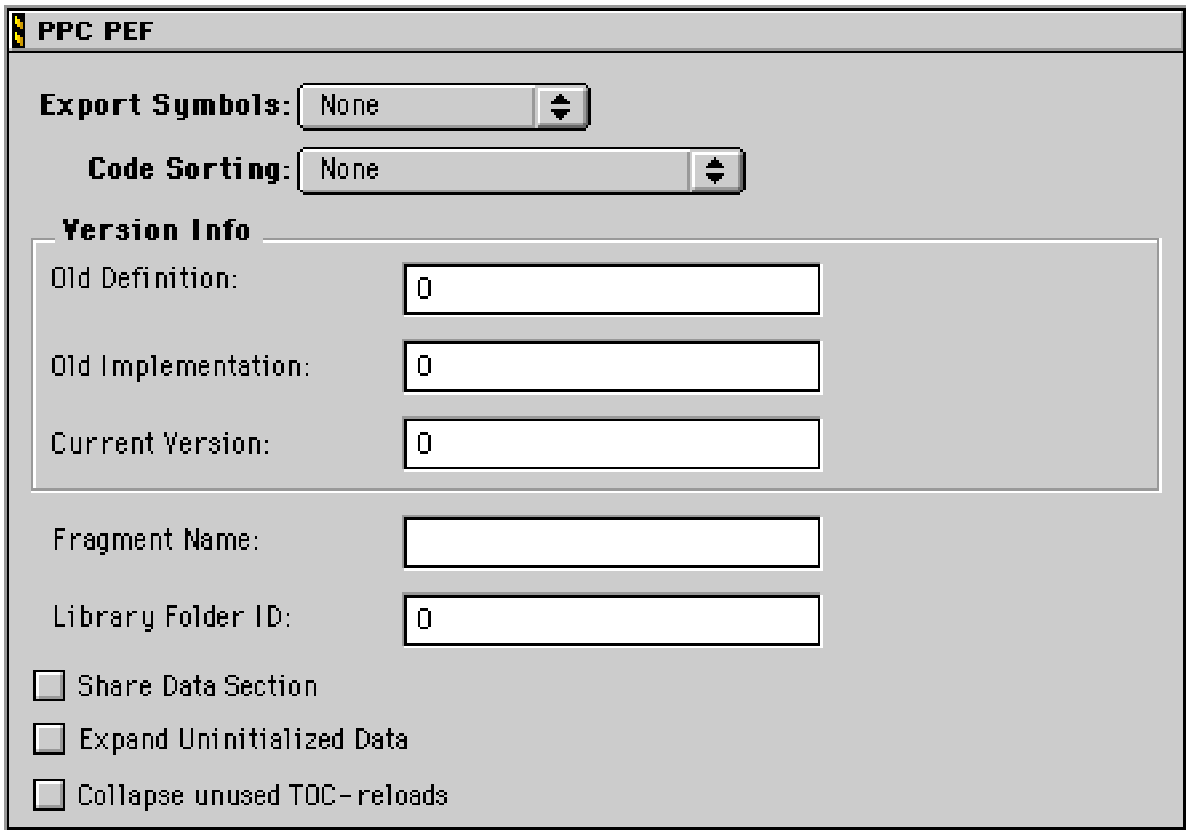
Termination

Specifies the symbol name of the routine that PowerPC will call when the fragment's data instantiation is released. Typically, the termination routine deallocates and releases the objects and services that were prepared by the initialization routine.

PPC PEF

The **PPC PEF** panel, shown in Figure 10.8, is where you control settings related to linking your object code into a code fragment.

Figure 10.9 The PPC PEF panel.



These items in this panel are:

Export Symbols	Old Definition
Code Sorting	Current Version
Old Implementation	Library Folder ID
Fragment Name	Share Data
Expand Uninitialized Data	Collapse Unused TOC-reloads

Export Symbols

Specifies where to look for a list of symbols to export from the code fragment. Table 10.5 lists the options in this pop-up menu.

Table 10.5 Symbol export options

Choosing this option…	Tells the linker to
None	export no symbols
Use ".exp" file	export the symbols listed in a file ending with ".exp"
All globals	export all global symbols
Use #pragma	export all symbols enclosed between #pragma export on and #pragma export off directives

Code Sorting

Code sorting can be used when producing a shared library or an application. Code sorting affects the order that the PowerPC linker puts symbols into the PEF code fragment. This option allows you to optimize the symbol order so that symbols that reference each other often are close to each other, thereby improving cache performance. Table 10.6 describes each option.

Table 10.6 Code Sorting options

Option	Meaning
None	Default order. It is basically the link order specified in the link order view of the project window, except that text files are put before non-text files.
By #pragma segment	puts together symbols in the same segment, as specified by `#pragma segment` directives. The order is undefined within a segment.

Option	Meaning
By depth-first traversal	Puts symbols in the order of a depth-first traversal of the call graph, starting from the entry points. For example, if routine `main()` calls routines `A()` and `B()`, and routine `A()` calls `C()` and `D()`, then the depth-first order would be `main()`, `A()`, `C()`, `D()`, `B()`.
by breadth-first traversal	Similar to By depth-first traversal, except the order is that of a breadth-first traversal. Given the same example, the breadth-first order would be `main()`, `A()`, `B()`, `C()`, `D()`
Use ".arr" file	Causes the linker to get the symbol order from a `.arr` file, which contains a list of symbol names in a syntax identical to the `.exp` file. The linker looks for a `.arr` file in the project, or if there's not one in the project, then it looks for one named `[projectname].arr` in the project directory. If the option is on and the linker can not find a `.arr` file, then it creates one for you with the symbol names in the default order. This option can be used with optimizers such as MrPlus.

Old Definition

Specifies the oldest acceptable version for a fragment which is called by this fragment.

Old Implementation

Specifies the old acceptable version number for a fragment which refers to an item in this fragment.

Current Version

Specifies the version of this fragment.

Fragment Name

Specifies the name of the PowerPC code fragment. Leave this field blank if you want the code fragment name to be the same as the current target's output name. See "PPC Target" on page 175 for information on specifying the output name.

Library Folder ID

Specifies the resource ID of the alias resource that specifies where to search for libraries referred to by this code fragment.

Expand Uninitialized Data

Specifies whether or not to expand uninitialized data structures to their full size in the PEF container. If this option is selected, uninitialized objects are stored in uncompressed format in the PEF container. If this option is deselected, uninitialized objects are expanded to their full size at runtime only.

Share Data

Specifies whether or not the fragment's data is instantiated once. If selected, all processes that refer to the fragment use a single instantiation of the fragment's data. If deselected, a new instantiation of the fragment's data is created for each process that refers to the fragment.

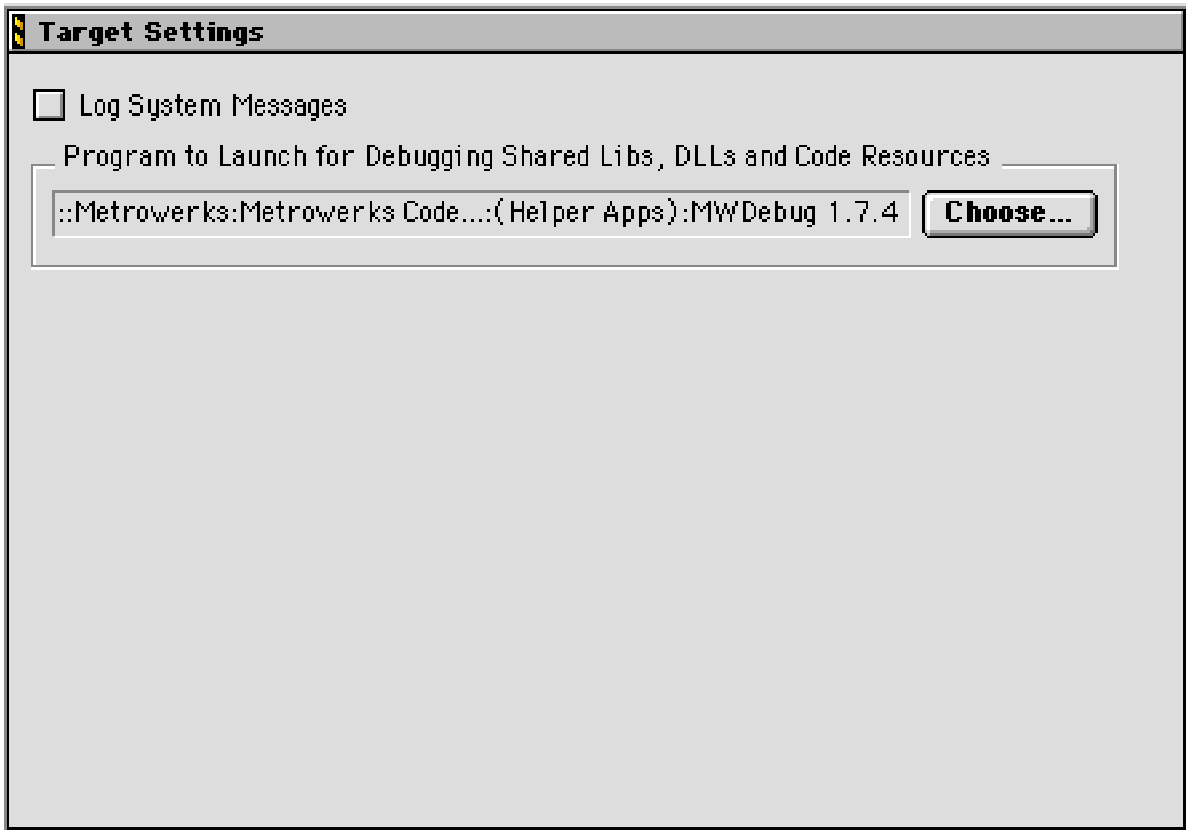
Collapse Unused TOC-reloads

Removes the unused no operation following a branch-and-link (BL) instruction if the BL doesn't go through glue code. This saves 4-bytes of space per BL, which actually turns out to be fairly significant for C++ applications, which make lots of calls.

Debugger Settings

The **Debugger Settings** panel presents options for selecting the tools to use for debugging your project.

Figure 10.10 Debugger Target Settings panel


Target Settings
Log System Messages
Program to Launch for Debugging Shared Libs, DLLs and Code Resources
::Metrowerks:Metrowerks Code...:(Helper Apps):MWDebug 1.7.4
Choose...

The options in this panel are:

Log System Messages

Program to Launch for Debugging

Log System Messages

Controls the display of messages in the debugger's log window. If selected, messages about the actions the debugger takes while controlling your application appear in the log window.

NOTE: This feature is only available when debugging on Windows.

Program to Launch for Debugging

Launches a program before running the project under the debugger's control. Click the **Choose** button to select a program. This option is most useful when debugging a piece of software that acts as a plugin or extension to another program.

11

C and C++ for PowerPC

This chapter describes both the CodeWarrior back-end compiler and linker for PowerPC.

NOTE: The “back-end” of the compiler refers to the module that actually generates code for the target processor. The “front-end” refers to the module that parses and interprets the source code.

The sections in this chapter are:

- Number Formats for PowerPC
- Calling Conventions for PowerPC
- Register Variables for PowerPC
- Optimizing Code for PowerPC
- Pragmas for PowerPC
- Linker Issues for PowerPC

However, this chapter does *not* discuss several important topics with respect to code generation, including front-end compiler issues, compiler and linker errors, controlling the size of C++ code, and so forth. These topics are covered in other CodeWarrior documentation as outlined in Table 11.1.

Table 11.1 Other compiler/linker documentation

For this topic...	See...
how CodeWarrior implements the C/C++ language	*C Compilers Reference* generally
using C/C++ Language and C/C++ Warnings settings panels	*C Compilers Reference*, "Setting C/C++ Compiler Options" chapter
controlling the size of C++ code	*C Compilers Reference*, "C++ and Embedded Systems" chapter
using compiler pragmas	*C Compilers Reference*, "Pragmas and Symbols" chapter
initiating a build, controlling which files are compiled, handling error reports	*IDE User Guide*, "Compiling and Linking" chapter
information about a particular error	*Error Reference*
PowerPC assembler	*Assembler Guide*

NOTE: Some of the items discussed in this chapter may actually be implemented in the front-end compiler. However, it really doesn't matter whether the actual implementation of a feature occurs in the front-end or back-end compiler. From the programmer's point of view, it is all one compiler.

Number Formats for PowerPC

This section describes how the CodeWarrior C/C++ compilers implement integer and floating-point types for PowerPC processors. You can also read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types.

The topics in this section are:

- PowerPC Integer Formats
- PowerPC Floating-Point Formats

PowerPC Integer Formats

Table 11.2 shows the size and range of the integer types for the PowerPC compiler.

Table 11.2 PowerPC Integer Types

For this type	Option setting	Size is	and its range is
bool	n/a	8 bits	`true` or `false`
`char`	Use Unsigned Chars is off (see language preferences panel in the "C Compilers Reference.")	8 bits	`-128` to `127`
	Use Unsigned Chars is on	8 bits	`0` to `255`
signed char	n/a	8 bits	`-128` to `127`
`unsigned char`	n/a	8 bits	`0` to `255`
`short`	n/a	16 bits	`-32,768` to `32,767`
`unsigned short`	n/a	16 bits	`0` to `65,535`
`int`	n/a	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned int`	n/a	32 bits	`0` to `4,294,967,295`
`long`	n/a	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned long`	n/a	32 bits	`0` to `4,294,967,295`

For this type	Option setting	Size is	and its range is
long long	n/a	64 bits	`-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`
unsigned long long	n/a	64 bits	`0` to `18,446,744,073,709,551,615`

PowerPC Floating-Point Formats

Table 11.3 shows the sizes and ranges of the floating point types for the PowerPC Mac OS compiler.

Table 11.3 PowerPC floating point types

Type	Size	Range
`float`	32 bits	`1.17549e-38` to `3.40282e+38`
short double	64 bits	`2.22507e-308` to `1.79769e+308`
double	64 bits	`2.22507e-308` to `1.79769e+308`
long double	64 bits	`2.22507e-308` to `1.79769e+308`

Calling Conventions for PowerPC

(K&R, §A8.6.3) This section describes the C/C++ calling conventions for PowerPC on MacOS.

The consortium that created the PowerPC processor created a standard set of calling conventions. The rest of this section describes how CodeWarrior PowerPC back-end compiler implements these standards. For more information on these calling conventions, see *Inside Macintosh: PowerPC System Software*.

The compiler reserves space for a function's parameters in two places: it reserves space for all parameter values in a structure in the caller's parameter area, and then it copies as many parameters as possible in registers. If the compiler copies a parameter into a register, it does not also copy it onto the parameter stack, but the compil-

er still reserves space for it on the stack. Placing parameters in registers avoids memory references to the parameter area and speeds up your programs.

In the parameter area, parameters are laid out in the order they appear, with the left-most parameter at the lowest offset. Each parameter starts at a word boundary regardless of size. For example, characters take up a word. Doubles may not be on a double word boundary, but are at least on a word boundary. Signed integers smaller than a word are sign-byte extended to a word. Unsigned chars are zero-extended.

In the registers, the compiler maps the first eight words of the parameter area — excluding floating point values — to the general purpose registers `r3` through `r10`. Integers and pointers take up one register each. Composite parameters (such as structs, classes, and arrays) take up as many consecutive registers as they need. Note that the compiler maps composite parameters to the registers as raw data, not as individual members or elements. For example, an array of six chars uses two registers: all of the first and the top half of the second register.

NOTE: A composite parameter may be passed both in registers and the parameter stack. If a parameter starts in or before the eighth word and ends after the eighth word, the compiler stores part of it in registers and the rest on the parameter stack.

Floating-point values are mapped to the floating point registers `fp1` through `fp13`. The compiler maps only free variables and not floating-point values contained in composite types. If the floating-point parameter appears within the first eight words, the compiler does not use the corresponding general register or pair of registers. The compiler does not use the register but simply skips it. The compiler does not skip floating-point registers but uses them consecutively.

If a function does not have a prototype, or has a variable argument list, the compiler copies the floating-point arguments into both general purpose registers and floating-point registers. In other words, the general purpose registers contain the first eight bytes of *all* pa-

rameters, and the floating-point registers contain duplicates of the floating-point parameters. The compiler performs this duplication because the function may be expecting either floats or raw data. If the function definition specifies floats, it will look for the parameters in the floating-point registers. If the function accepts anything and interprets the raw data itself (like `printf()`), it will look for the parameters in the general purpose registers.

For the function `foo()`, shown below, note that `r4`, `r5`, and `r6` are empty and that the floating-point members of the struct are not stored in floating-point registers. Also, the compiler fills up the floating-point registers one after the other, even though the floating-point parameters do not follow each other.

```
typedef struct rec {
  int i;
  float f;
  double d;
} rec;

void foo( int i1, float f1, double d1, rec r,
          int i3, float f3, double d3 );
```

This list describes where the compiler places a return value:

- It returns integer values in `r3`.
- It returns float and double floating-point values in `fp1`.
- If it returns a composite type (such as a struct, class, or array), it allocates area for the return value in a temporary storage area, and returns a pointer to that area as an implicit left-most parameter (that is, in `r3`).

Register Variables for PowerPC

(K&R, §A4.1, §A8.1) The PowerPC back-end compiler automatically allocates local variables and parameters to registers according to how frequently they're used and how many registers are available. If you're optimizing for speed, the compiler give preference to variables used in loops.

The PowerPC back-end compiler gives preference to variables declared to be `register`, but does not automatically assign them to registers. For example, the compiler is more likely to place a variable from an inner loop in a register than a variable declared register.

The PowerPC back-end compiler can use these registers for local variables:

- `GPR13` through `GPR31` for integers and pointers
- `FPR14` through `FPR31` for floating point variables.

Optimizing Code for PowerPC

This section discusses optimizations that are specific to PowerPC development with CodeWarrior. These are:

- Register Coloring

More optimizations are available through the **Global Optimizations** panel. For information about these optimizations, see the *IDE User Guide*.

Register Coloring

The PowerPC back-end compiler can perform a register optimization called *register coloring*. In this optimization, the compiler lets two or more variables share a register: it assigns different variables or parameters to the same register if you do not use the variables at the same time. In this example, the compiler could place `i` and `j` in the same register:

```
short i;
int j;

for (i=0; i<100;  i++) {   MyFunc(i);  }
for (j=0; j<1000; j++) {   OurFunc(j); }
```

However, if a line like the one below appears anywhere in the function, the compiler would realize that you're using `i` and `j` at the same time and place them in different registers:

```
int k = i + j;
```

You control whether the PPC back-end compiler performs the register coloring optimization from the **Global Optimizations** settings panel. Refer to the *IDE User Guide* for information on this panel.

If register coloring is on while you debug your project, it may appear as though there's something wrong with the variables sharing a register. In the example above, `i` and `j` would always have the same value. When `i` changes, `j` changes in the same way. When `j` changes, `i` changes in the same way. To avoid this confusion while debugging, turn off register coloring or declare the variables you want to watch as volatile.

Pragmas for PowerPC

Table Table 11.4 list the pragmas supported for PowerPC development.

Table 11.4 Pragmas for PowerPC development

align	align_array_members	ANSI_strict	ARM_conform
auto_inline	bool	check_header_flags	cplusplus
cpp_extensions	dont_inline	dont_reuse_strings	enumsalwaysints
exceptions	extended_errorcheck	fp_contract	global_optimizer
ignore_oldstyle	inline_depth	longlong	longlong_enums

mark	once	only_std_keywords	optimize_for_size
peephole	pop	precompile_target	push
readonly_strings	require_prototypes	RTTI	scheduling
static_inlines	syspath_once	trigraphs	unsigned_char
unused	wchar_type	warn_emptydecl	warning_errors
warn_extracomma	warn_hidevirtual	warn_illpragma	warn_implicitconv
warn_possunwant	warn_unusedarg	warn_unusedvar	

Table 11.5 lists additional pragmas available only for embedded (ELF/DWARF) development.

Table 11.5 Pragmas for PowerPC MacOS development

direct_to_SOM	disable_registers	export	import
internal	libexport	mpwc_newline	mpwc_relax
profile	segment	SOMCall Optimization	SOMCallStyle
SOMCheck Environment	SOMClassVersion	SOMMetaClass	SOMReleaseOrder
toc_data	traceback		

These pragmas, pragma syntax, and how to determine and modify the state of the compiler using pragmas are all detailed in the *C Compilers Reference*.

Linker Issues for PowerPC

This section discusses the background information on the PowerPC linker and how it works. The topics in this section are:

- Deadstripping Unused Code and Data
- Link Order

Deadstripping Unused Code and Data

The PowerPC linker deadstrips unused code and data only from files compiled by the CodeWarrior C/C++ compiler. Assembler relocatable files and C/C++ object files built by other compilers are never deadstripped. Deadstripping is particularly useful for C++ programs. Libraries (archives) built with the CodeWarrior C/C++ compiler only contribute the used objects to the linked program. If a library has assembly or other C/C++ compiler built files, only those files that have at least one referenced object contribute to the linked program. Completely unreferenced object files are always ignored.

Link Order

Link order is generally specified in the Link Order view of the Project window. For general information on setting link order, see the *IDE User Guide*.

Regardless of the link order specified in the Link Order view of the Project window, the PowerPC linker always processes C/C++ and assembler source files before it processes object files (.o) and archive files (.a), which are treated as libraries. Therefore if a symbol is defined in a source file, the linker will use that definition in preference to a definition in a library.

12

Pascal for PowerPC

This section contains topics for using the CodeWarrior Pascal compiler for PowerPC processors.

PowerPC Notes Overview

This chapter contains information specific to creating a program targeting PowerPC processors with CodeWarrior Pascal. The remainder of it covers the following topics:

- Data Type Formats for PowerPC
- Calling Conventions for PowerPC

Data Type Formats for PowerPC

This section describes how the CodeWarrior Pascal compiler implements classes of standard Pascal data types when targeting the PowerPC platform. These classes are:

- Integer Types for PowerPC
- Floating Point Types for PowerPC
- Array Types and PACKED Arrays for PowerPC
- Character String Types for PowerPC
- Sets for PowerPC
- Enumerations for PowerPC
- Records for PowerPC

Integer Types for PowerPC

This section describes how INTEGER, LONGINT and a few other important atomic data types are implemented for PowerPC in CodeWarrior Pascal.

`INTEGER` for PowerPC

On the PowerPC platform these are 16-bit, 2's-complement integers. The range of the type `INTEGER` is the set of values

```
-(MaxINT+1), -MaxINT,...,
-1, 0, 1, ... MaxINT-1, MaxINT
```

that is, –32768 to +32767.

`LONGINT` for PowerPC

For the PowerPC platform these are 32-bit, 2's-complement integers. The range of the type `LONGINT` is the set of values

```
-(MaxLONGINT+1), -MaxLONGINT, ...-1, 0, 1,
... MaxLONGINT-1, MaxLONGINT
```

that is –2147483648 to +2147483647.

Arithmetic on `INTEGER` and `LONGINT` operands is done in both 16-bit and 32-bit precision, as follows:

- All integer constants in the range of type `INTEGER` are considered to be of type `INTEGER`. All integer constants in the range of type `LONGINT`, but not in the range of type `INTEGER`, are considered to be of type `LONGINT`.
- When both operands of an operator (or the single operand of a unary operator) are of type `INTEGER`, 16-bit operations are always performed and the result is of type `INTEGER` (truncated to 16 bits if necessary). Similarly, if both operands are of type `LONGINT`, 32-bit operations are always performed and the result is of type `LONGINT`.
- When one operand is of type `LONGINT` and the other is of type `INTEGER`, the `INTEGER` operand is converted to `LONGINT`, 32-bit operations are performed, and the result is of type `LONGINT`. However, if this value is assigned to a variable of type `INTEGER`, it is truncated (see next rule).

- The expression on the right of an assignment statement is evaluated independently of the size of the variable on the left. For example, if variable `longVar` is declared as type `LONGINT`, the statement `longVar := MaxINT+MaxINT` will still cause `INTEGER` overflow. If necessary, the result of the expression is truncated or extended to match the size of the variable on the left.

An important point to remember is that each operator is applied only to its two operands, so, at most, one of those operands is converted. If the expression contains other operands, those are not necessarily converted. For example, in the expression

```
oneInt+twoInt+threeInt+oneLongint
```

the value of `oneInt` is added to `twoInt` in a 16-bit operation, the result is added to `threeInt` in another 16-bit operation, and the result of that is converted to a `LONGINT` value and added to `oneLongint`. The result of the expression is a `LONGINT`.

The `Ord4` function described in the Pascal Library Reference can be used to convert an `INTEGER` value to a `LONGINT` value.

`UNSIGNEDBYTE` for PowerPC

The CodeWarrior Pascal compiler implements this type as an 8-bit unsigned integer for the PowerPC platform. The range of the type `UNSIGNEDBYTE` is the set of values 0 to +255.

This type can be used in expressions with the `UNSIGNEDWORD`, `UNSIGNEDLONG`, `INTEGER` and `LONGINT` types, but no range checking is available.

`UNSIGNEDWORD` for PowerPC

The CodeWarrior Pascal compiler implements this type as a 16-bit unsigned integer for the PowerPC platform. The range of the type `UNSIGNEDWORD` is the set of values

```
0, 1, ..., 2 * MaxINT, (2 * MaxINT ) + 1
```

that is, 0 to +65535.

This type can be used in expressions with the `UNSIGNEDLONG`, `INTEGER` and `LONGINT` types, but no range checking is available.

`UNSIGNEDLONG` for PowerPC

The CodeWarrior Pascal compiler implements this type as a 32-bit unsigned integer for the PowerPC platform. The range of the type `UNSIGNEDLONG` is the set of values

```
0, 1, ..., 2 * MaxLONGINT, (2 * MaxLONGINT ) + 1
```

that is, 0 to +4294967295.

This type can be used in expressions with the `UNSIGNEDWORD`, `INTEGER` and `LONGINT` types, but no range checking is available.

NOTE: For PowerPC, the `UNSIGNEDLONG` type is equivalent to PACKED ARRAY [0 ..3] OF CHAR.

This is necessary because of Apple's decision, when Mac OS ran only on 68K processors, to allow arrays of 4 characters to be accessible both as arrays of 4 characters and as unsigned long integers. These could be directly pushed onto and popped from the stack, rather than the standard method of their addresses being pushed and popped.

With the arrival of PowerPC, however, Mac OS changed to pass all arrays by address, out of necessity.

Floating Point Types for PowerPC

There are four floating-point types in CodeWarrior Pascal, all predefined:

- `REAL` for PowerPC
- `SINGLE` for PowerPC
- `DOUBLE` for PowerPC
- `LONGREAL` for PowerPC

The ANS `REAL for PowerPC` type is identical to the `SINGLE for PowerPC` type. CodeWarrior Pascal accepts both identifiers and treats them identically.

REAL for PowerPC

For PowerPC, this type takes up four bytes of storage (32 bits). The range of this type is approximately 1.401298464E–45 to 3.402823466E38 in scientific notation. `REAL` has 7 or 8 digits of precision.

SINGLE for PowerPC

This type is the same as `REAL` for PowerPC.

LONGREAL for PowerPC

For PowerPC, unlike 68K, in CodeWarrior Pascal this type has a fixed length: 8 bytes (64 bits).

DOUBLE for PowerPC

This type is the same as `LONGREAL` for PowerPC.

128-Bit Floating Point Numbers, PowerPC, and CW Pascal

Experienced PowerPC programmers may be aware that PPC processors have the built-in ability to handle 128-bit (16-byte) floating-point numbers.

It's true that you can take advantage of this ability through the inline assembler built in to the CodeWarrior Pascal compiler, or in object code created externally and linked in at compile time.

However, all floating-point symbols larger than 8 bytes will be truncated to a size of 8 bytes at compile time. Remember this if you use public floating-point symbols greater than 8 bytes in size in inline assembler or externally-created object code.

Array Types and PACKED Arrays for PowerPC

This section documents how array and PACKED array types are implemented for PowerPC in CodeWarrior Pascal.

Array Types for PowerPC

For the PowerPC platform, CodeWarrior Pascal implements arrays using even padded elements. That is, the compiler ensures that each array element begins at an address or offset that is divisible by 2.

PACKED Arrays for PowerPC

PACKED arrays on PowerPC are implemented like standard arrays, using even padded elements, but the addresses or offsets will not necessarily be divisible by 2 depending on what you have selected for the **Struct Alignment** option in your PowerPC project. The divisor will be:

- 2 if **Struct Alignment** is set to **68K**
- 4 if **Struct Alignment** is set to **68K 4-byte**
- 4 or 8 if **Struct Alignment** is set to **PowerPC**, depending on the size of the array elements. The compiler makes the determination on-the-fly.

Also, PACKED arrays of characters use 8 bits per character for storage, not 16 bits.

Character String Types for PowerPC

This section describes how character string types are implemented for PowerPC in CodeWarrior Pascal.

Pascal's STRING for PowerPC

For PowerPC, a Pascal string is stored as a one-byte-length field followed by the characters in the string. You can therefore change the length of the string by changing its zeroth character. For example,

```
myStr[0] := Chr(Ord(myStr[0])+n);
```

changes the length of `myStr` by the value of `n`.

C and PACKED ARRAY OF CHAR for PowerPC

In CodeWarrior Pascal, when you declare a string constant using double quotation marks (`"`) rather than Pascal-standard single quotation marks (`'`), that string is stored as a C-style string. That means the first byte of the string does not contain the string's length, and it is terminated with an ASCII 0 (NUL) character.

Care is required when assigning C-style string constants to other variables and passing them in procedures and functions:

- If you want `destination` in

  ```
  destination := CStyleStringConstant
  ```

  to also be stored as a C-style string you must declare it as a PACKED array of char. This is also true when `destination` is a procedure or function parameter to which the value of a C-style string constant will be assigned.

  If `destination` is declared as an ordinary Pascal string, after assignment it will contain the standard prefixing length byte and the first 255 bytes of the C-style string, and the terminating NUL will be removed.

- The size in bytes of the `PACKED` array of char you use for `destination` must be exactly equal to the length of the C-style string constant including the terminating NUL, not shorter and not longer. If

  ```
  const CStringConstant = "hello"
  ```

  and you will assign the value of `CStringConstant` to `destination`, then

  ```
  var
    destination: PACKED array[6] of char
  ```

Sets for PowerPC

This section documents anything distinctive about how sets are implemented for PowerPC in CodeWarrior Pascal.

Isolating one member of a set for PowerPC

An individual member of a set declared in CodeWarrior Pascal can be isolated using the following formula:

```
bit : BAND(SET[|SET| - 1 - EL div 8],(BSL(1,EL mod 8))
```

Enumerations for PowerPC

This section documents anything distinctive about how enumerations are implemented for PowerPC in CodeWarrior Pascal.

Storage of enumerated constants for PowerPC

Enumerated constants are each allocated one or two bytes of storage depending on the number of constants in the enumeration:

- One byte each if the enumeration contains 256 constants or fewer
- Two bytes each if the enumeration contains more than 256 constants

Records for PowerPC

This section documents anything distinctive about how records are implemented for PowerPC in CodeWarrior Pascal.

Record field alignment for PowerPC

Record fields are aligned on boundaries of:

- 2 bytes if **Struct Alignment** is set to **68K** in your 68K project
- 4 bytes if **Struct Alignment** is set to **68K 4-byte** or it's set to PowerPC and the **IEEE doubles** option is unselected.
- 8 bytes if **Struct Alignment** is set to **PowerPC** and the **IEEE doubles** option is selected.

Calling Conventions for PowerPC

For programs targeting the PowerPC platform, the Metrowerks Pascal compiler follows a standard set of calling conventions dictated by the consortium behind the PowerPC. For more information on these calling conventions, see *Inside Macintosh: PowerPC System Software*.

The topics in this section are:

- Overview of Calling Conventions for PowerPC
- Parameters and PowerPC Calling Conventions
- Registers and PowerPC Calling Conventions
- Floating-point Values and PowerPC Calling Conventions
- Functions and PowerPC Calling Conventions

Overview of Calling Conventions for PowerPC

The compiler reserves space for a function's parameters in two places: it reserves space for all parameter values in a structure in the caller's parameter area, and then it copies as many parameters as possible in registers.

If the compiler copies a parameter into a register, it does not also copy it onto the parameter stack, but the compiler still reserves space for it on the stack. Placing parameters in registers avoids memory references to the parameter area and speeds up your programs.

NOTE: A *word* is four bytes on the PowerPC.

Parameters and PowerPC Calling Conventions

In the parameter area, parameters are laid out in the order they appear, with the left-most parameter at the lowest offset. Each parameter starts at a word boundary regardless of size.

For example, characters take up a word and doubles may not be on a double word boundary. Signed integers smaller than a word are sign-byte extended to a word. Unsigned chars are zero-extended.

Registers and PowerPC Calling Conventions

In the registers, the compiler maps the first eight words of the parameter area — excluding floating point values — to the general purpose registers `r3` through `r10`. Integers and pointers take up one register each. Composite parameters (such as structs, classes, and arrays) take up as many consecutive registers as they need.

Note that the compiler maps composite parameters are to the registers as raw data, not as individual members or elements. For example, an array of six chars uses two registers: all of the first and the top half of the second.

NOTE: A composite parameter may be both in registers and the parameter stack. If the parameter starts in or before the eighth word and ends after the eighth word, the compiler stores part of it in registers and the rest on the parameter stack.

Floating-point Values and PowerPC Calling Conventions

Floating-point values are mapped to the floating point registers `fp1` through `fp13`. The compiler maps only free variables and not floating-point values contained in composite types.

If the floating-point parameter appears within the first eight words, the compiler does not use the corresponding general register or pair of registers. The compiler does not use the register but simply skips it. The compiler does not skip floating-point registers but uses them consecutively.

Functions and PowerPC Calling Conventions

If a function has a variable argument list, the compiler copies the floating-point arguments into both general purpose registers and floating-point registers. In other words, the general purpose registers contain the first eight bytes of *all* parameters, and the floating-point registers contain duplicates of the floating-point parameters.

The compiler performs this duplication since the function may be expecting either floats or raw data. If the function definition specifies floats, it will look for the parameters in the floating-point registers. If the function accepts anything and interprets the data itself (like `printf()`), it will look for the parameters in the general purpose registers.

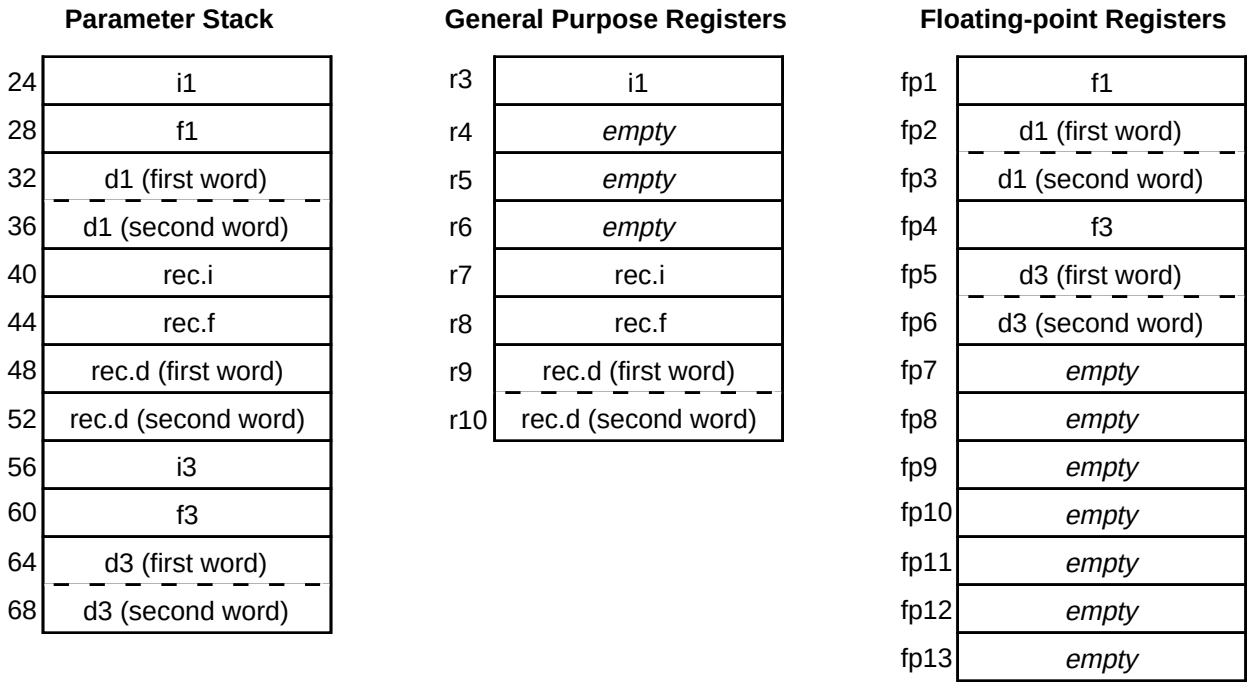
PowerPC Calling Conventions Example

The code fragment below shows how the compiler would store the parameters in routine `foo`. Note that `r4`, `r5`, and `r6` are empty and that the floating-point members of the struct are not stored in floating-point registers. Also, the compiler fills up the floating-point registers one after the other, even though the floating-point parameters do not follow each other.

```
type
  rec = record
    i: longint;
    f: real;
    d: double;
  end;

procedure foo (
  i1: longint;
  fl: real;
  d1: double;
  r: rec;
  i3:longint;
  f3:real;
  d3:double);
```

Figure 12.1 PowerPC parameter passing example



The compiler places return values as follows:

- It returns integer values in `r3`.
- It returns float and double floating-point values in `fp1`.
- If it returns a composite type such as a record or array, it allocates area for the return value in a temporary storage area, and returns a pointer to that area as an implicit left-most parameter (that is, in `r3`).

13

Inline PowerPC Assembler

This chapter describes support for inline assembly language programming built into the CodeWarrior compilers.

This chapter does *not* discusses the stand-alone assembler available for PowerPC. For information on the stand-alone assembler, see the *Assembler Guide*.

This chapter does not document all the instructions available in PowerPC assembly language. For information on PowerPC assembly language instructions, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors*, published by Motorola.

The sections in this chapter are:

- Working With Inline Assembly for PowerPC
- Assembler Directives for PowerPC
- Intrinsic Functions for PowerPC

Working With Inline Assembly for PowerPC

This section describes how to use the CodeWarrior compiler's built-in support for assembly language programming, including assembler syntax.

The topics in this section are:

- Inline Assembler Syntax for PowerPC
- Special PowerPC Instructions
- Defining an Assembly Function for PowerPC

- Creating Labels for PowerPC Assembly
- Using Comments in PowerPC Assembly
- Using the Preprocessor in PowerPC Assembly
- Using Local Variables and Arguments in PowerPC Assembly
- Creating a Stack Frame in PowerPC Assembly
- Specifying Operands in PowerPC Assembly

Inline Assembler Syntax for PowerPC

To specify that a block of code in your file should be interpreted as assembly language, use the `asm` keyword.

To ensure that the C/C++ compiler recognizes the `asm` keyword, you must turn off the ANSI Keywords Only option in the C/C++ Language panel. This panel and its options are fully described in the *C Compilers Reference*.

The assembly instructions are the standard PowerPC instruction mnemonics. For information on PowerPC assembly language instructions, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors*, published by Motorola (serial number MPCFPE32B/AD).

The Apple *Assembler for PowerPC* for the MPW `PPCAsm` assembler is also a good reference and is on the CodeWarrior CD.

You use the PowerPC inline assembler to specify that an *entire function* is in assembly language. Assembly statement blocks within a function are not supported. In other words, the PowerPC inline assembler is a function-level assembler.

TIP: To enter a few lines of assembly language code within a single function, you can use the compiler's support for intrinsics. See "Intrinsic Functions for PowerPC" on page 243.

Inline assembly code for PowerPC uses the following syntax:

```
asm function definition
```

For example,

```
asm long MyFunc(void)  // OK, an assembly function
{
  . . . // assembly instructions
}
```

However, the following code would not be legal:

```
long MyFunc (void)
{
  asm {. . .}  // Error, assembly statement blocks not supported
}
```

NOTE: Assembly language functions are never optimized, regardless of compiler settings.

Special PowerPC Instructions

To set the branch prediction (y) bit for those branch instructions that can use it, use + or -. For example:

```
@1  bne+   @2    // Predicts branch taken
@2  bne-   @1    // Predicts branch not taken
```

Most integer instructions have four different forms:

- normal
- record, which sets register cr0 to whether the result is less, than, equal to, or greater than zero. This form ends in a period (".").
- overflow, which sets the SO and OV bits in the XER if the result overflows. This form ends in the letter "o".

- overflow and record, which sets both registers. This form ends in "`o.`".

```
add   r3,r4,r5 //  Normal add
add.  r3,r4,r5 //  Add with record: sets cr0
addo  r3,r4,r5 //  Add with overflow:sets XER
addo. r3,r4,r5 //  Add with overflow and record: sets cr0 and XER
```

Some instructions only have a record form (with a period). Make sure to include the period always:

```
andi.  r3,r4,7   //  '.' is not optional here
andis. r3,r4,7   //  Or here
stwcx. r3,r4,r5  //  Or here
```

Defining an Assembly Function for PowerPC

To include assembly in your PowerPC project, declare a function with the `asm` qualifier, like this:

```
asm long f(void) { . . . } // OK: An assembly function
```

Note that you cannot create an assembly statement block within a C function:

```
long f(void)
{
  asm { . . . }    // ERROR: Assembly statement
}                  // blocks are not supported.
```

The built-in assembler uses all the standard PowerPC assembler instructions. It accepts some additional directives described in "Assembler Directives for PowerPC" on page 240. If you use the `machine` directive, you can also use instructions that are available only in certain versions of the PowerPC. For more information, see "machine" on page 242.

Keep these tips in mind as you write assembly functions:

- All statements must follow this syntax:

```
[LocalLabel:] (instruction | directive) [operands]
```

  Each instruction must end with a newline or a semicolon (`;`).
- Hex constants must be in C-style, not Pascal-style. For example:

```
li    r3, 0xABCDEF      // OK
li    r3, $ABCDEF       // ERROR
```

- Assembler directives, instructions, and registers are case-sensitive and must be in lowercase. For example these two statements are different:

```
add   r2,r3,r4          // OK
ADD   R2,R3,R4          // ERROR
```

- Every assembly function must end in an `blr` statement. The compiler does not add one for you. For example:

```
asm void f(void)
{
    add  r2,r3,r4
}                   // ERROR: No blr statement

asm void g(void)
{
    add  r2,r3,r4
    blr             // OK
}
```

Listing 13.1 shows an example of an assembly function.

Listing 13.1 Creating an assembly function

```
asm void mystrcpy(char *tostr, char *fromstr)
{
    addi   tostr,tostr,-1
```

```
    addi   fromstr,fromstr,-1
@1  lbzu   r5,1(fromstr)
    cmpwi  r5,0
    stbu   r5,1(tostr)
    bne    @1
    blr
}
```

Creating Labels for PowerPC Assembly

A label can be any identifier that you haven't already declared as a local variable. The name may start with `@`, so these are legal names: `foo`, `@foo`, and `@1`. Only labels that don't start with `@` need to end in a colon. For example:

```
asm void foo(void)
{
x1:  add    r3,r4,r5      // OK, has colon
@x2: add    r6,r7,r8      // OK, has both @ and colon
x3   add    r9,r10,r11    // ERROR, Needs colon
@x4  add    r12,r13,r14   // OK, starts with @
}
```

NOTE: The first statement in an assembly function cannot be a label that starts with @.

Using Comments in PowerPC Assembly

You cannot begin comments with a pound sign (#), since the preprocessor uses the pound sign. However, you can use C and C++ comments. For example:

```
add   r3,r4,r5    # ERROR
add   r3,r4,r5    // OK
add   r3,r4,r5    /* OK */
```

Using the Preprocessor in PowerPC Assembly

You can use all preprocessor features, such as comments and macros, in the assembler. However you must end each assembly statement with a semicolon (`;`), because the preprocessor ignores newlines. For example:

```
#define remainder(x,y,z)  \
divw    z,x,y; \
mullw   z,z,y; \
subf    z,z,x
```

Using Local Variables and Arguments in PowerPC Assembly

To refer to a memory location, you can use the name of a local variable or argument.

NOTE: You can refer to local variables by name even if a function does not contain the `fralloc` directive.

The rule for assigning arguments to registers or memory depends on whether the function has a stack frame. If function has a stack frame, the in-line assembler assigns:

- scalar arguments declared `register` to `r13-r31`
- floating-point arguments declared `register` to `fp14-fp31`
- other arguments to memory locations
- scalar locals declared `register` to `r13-r31`
- floating-point locals declared `register` to `fp14-fp31`
- other locals to memory locations

If function has no stack frame, the in-line assembler assigns:

- arguments that are declared `register` and passed in registers to the appropriate register
- other arguments to memory locations

- all locals to memory locations

NOTE: If there is no stack frame, a function cannot have more than 224 bytes of local variables.

For more information on PowerPC register conventions and argument-passing conventions, see "C and C++ for PowerPC" on page 207.

Creating a Stack Frame in PowerPC Assembly

You need to create a stack frame for a function, if the function:

- calls other functions
- uses more than 224 bytes of local variables
- declares local register variables.

The easiest way to create a stack frame is to use the `fralloc` directive at the beginning of your function and the `frfree` directive just before the `blr` statement. The `fralloc` directive automatically allocates and de-allocates memory for local variables and saves and restores the register contents.

```
asm void foo ()
{
  fralloc
  // Your code here
  frfree
  blr
}
```

The `fralloc` directive has an optional argument *number* which lets you specify the size in bytes of the parameter area of the stack frame. By default, the compiler creates a 32-byte parameter area. If your assembly-language routine calls any function that takes more than 32 bytes of parameters, you must specify a larger amount.

Specifying Operands in PowerPC Assembly

This topic describes how to specify the operands for assembly language instructions. It discusses how to use registers, labels, variable names, and immediate operands.

Using registers for PowerPC assembly

For a register operand, you must use one of the register names of the appropriate kind for the instruction. The register names are case-sensitive. You can also use a symbolic name for an argument or local variable that was assigned to a register.

The general registers are `RTOC`, `SP`, `r0` to `r31`, and `gpr0` to `gpr31`. The floating-point registers are `fp0` to `fp31 and f0` to `f31`. The condition registers are `cr0` to `cr7`.

Using labels for PowerPC assembly

For a label operand, you can use the name of a label. For long branches (such as `b` and `bl` instructions) you can also use function names. For `bla` and `la` instructions, you use absolute addresses. For other branches, you must use the name of a label. For example:

```
b    @3    // OK: Branch to local label
b    foo   // OK: Branch to external function foo
bl   @3    // OK: Call local label
bl   foo   // OK: Call external function foo
bne  foo   // ERROR: Short branch outside function
```

Using variable names as memory locations for PowerPC assembly

Whenever an instruction requires a memory location (such as a load instruction, a store instruction, or `la`), you can use a local or global variable name. You can modify local variable names with struct member references, class member references, array subscripts, or constant displacements. For example, all of the following are valid local variable references:

```
lwz  r3,myVar(SP)  // load myVar into r3
la   r3,myVar(SP)  // load address of myVar into r3
lwz  r3,myRect.top
lwz  r3,myArray[2](SP)
lwz  r3,myRectArray[2].top
lbz  r3,myRectArray[2].top+1(SP)
```

You can also use a register variable that is a pointer to a struct or class to access a member of the struct. For example:

```
register Rect *p;
lwz  r3,p->top;
```

Global variable names always refer to the TOC pointer for the variable, not to the variable itself, so you cannot modify them:

```
lwz  r3,myGlobalRect(RTOC) // load TOC pointer for myGlobalRect
lwz  r4,Rect.top(r3)           // fetch 'top' field
lwz  r3,myGlobalRect.top(RTOC) // nonsensical
```

You use the same method for obtaining the address of a function:

```
lwz  r3,myFunction(RTOC)
     // load TOC-pointer for TVector to myFunction
```

You can use the `@hiword` and `@loword` directives to access parts of a variable defined `long long`:

```
long long gTheLongLong = 5;

void Foo(void);
void Foo(void)
{
  fralloc

  lwz r5, gTheLongLong@hiword  // the upper word of gTheLongLong
  lwz r6, gTheLongLong@loword  // the lower word of gTheLongLong
```

```
  frfree
  blr
}
```

Using immediate operands for PowerPC assembly

For an immediate operand, you can use an integer or enum constant, `sizeof` expression, and any constant expression using any of the C dyadic and monadic arithmetic operators. These expressions follow the same precedence and associativity rules as normal C expressions. The in-line assembler carries out all arithmetic with 32-bit signed integers.

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```
lwz   r4,Rect.top(r3)
addi  r6,r6,Rect.left
```

You can also use the top or bottom half-word of an immediate word value as an immediate operand. To do this, use one of the @ modifiers, as illustrated below:

```
long gTheLong;

asm void foo(void)
{
  fralloc

  ori  r6, gTheLong@ha # upper halfword of address of "gTheLong"
  ori  r7, gTheLong@h  # upper halfword of address of "gTheLong"
  addi r7, gTheLong@l  # lower halfword of address of "gTheLong"

  frfree
  blr
}
```

Assembler Directives for PowerPC

This section describes some special assembler directives that the PowerPC built-in assembler accepts. They are:

- entry
- fralloc
- frfree
- machine

entry

```
entry [ extern | static ] name
```

PowerPC assembler directive that defines an entry point into the current function. Use the `extern` qualifier to declare a global entry point and use the `static` qualifier to declare a local entry point. If you leave out the qualifier, `extern` is assumed.

Listing 13.2 Using the entry directive

```
void __save_fpr_15(void);
void __save_fpr_16(void);
asm void __save_fpr_14(void)
{
    stfd   fp14,-144(SP)
    entry  __save_fpr_15
    stfd   fp15,-136(SP)
    entry  __save_fpr_16
    stfd  fp16,-128(SP)
    // ...
}
```

fralloc

```
fralloc [ number ]
```

PowerPC assembler directive that creates a stack frame for a function and reserves registers for your local register variables. You need to create a stack frame, if the function

- Calls other functions
- Uses more than 224 bytes of local variables
- Declares local register variables.

For more information, see "Creating a Stack Frame in PowerPC Assembly" on page 236.

The `fralloc` directive has an optional argument *number* which lets you specify the size in bytes of the parameter area of the stack frame. By default, the compiler creates a 32-byte parameter area. If your assembly-language routine calls any function that takes more than 32 bytes of parameters, you must specify a larger amount.

frfree

```
frfree
```

PowerPC assembler directive that frees the stack frame and restores the registers that `fralloc` reserved. For more information, see "Creating a Stack Frame in PowerPC Assembly" on page 236.

NOTE: The `frfree` directive does not generate a `blr` instruction. You must include one explicitly.

---

machine

`machine` *number*

PowerPC assembler directive that specifies which CPU the assembly code is for. The *number* must be one of the following:

```
601          603          604          all
```

If you use `all`, you can use only those instructions that are available on all PowerPC CPUs. If you don't use the `machine` directive, the compiler assumes `all`.

If you use `601`, you can also use the following instructions:

```
abs       abs.      abso      abso.     clcs
div       div.      divo      divo.     doz
doz.      dozo      dozo.     dozi      lscbx
lscbx.    maskg     maskg.    markir    markir.
mul       mul.      mulo      mulo.     nabs
nabs.     nabso     nabso.    rlmi      rlmi.
rrib      rrib.     sle       sle.      sleq
sleq.     sliq      sliq.     slliq     slliq.
sllq      sllq.     slq       slq.      sraig
sraig.    sraq      sraq.     sre       sre.
srea      srea.     sreq      sreq.     sriq
sriq.     srliq     srliq.    srlq      srlq.
srq       srq.      tlbie
```

If you use `603` or `604`, you can also use the following instructions:

```
fres      fres.     frsqrte   frsqrte.  fsel
fsel.     mftb      mftbl     stfiwx    tlbld
tlbli     tlbsync
```

smclass

```
smclass PR | GL
```

PowerPC assembler directive that lets you set the class for a function. By default, all functions have class {PR} which means they are normal executable code. If you're writing a glue routine, like the `__ptr_glue` routine that implements calls through function pointers, use `smclass GL` to set the class to {GL}.

You shouldn't need this directive for your own code, but CodeWarrior PowerPC runtime library uses it frequently.

Intrinsic Functions for PowerPC

This section discusses support for intrinsic functions in the CodeWarrior compilers. Support for intrinsic functions is not part of the ANSI C or C++ standards. They are an extension provided by the CodeWarrior compilers.

Intrinsic functions are a mechanism you can use to get assembly language into your source code. Here's how intrinsic functions work.

There is an intrinsic function for each processor opcode (instruction). Rather than using inline assembly syntax and specifying the opcode in an `asm` block, you call the intrinsic function that matches the opcode.

When the compiler encounters the intrinsic function call in your source code, it doesn't actually make a function call. The compiler substitutes the assembly instruction that matches your function call. As a result, no function call occurs in the final object code. The final

code is the assembly language instructions that correspond to the intrinsic functions.

TIP: Use intrinsic functions to add a few lines of assembly code within a function. If you want to write an entire function in assembly, you can use the inline assembler. See "Working With Inline Assembly for PowerPC" on page 229.

This section does not discuss every single intrinsic function. For information on PowerPC assembly language instructions, see *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors,* published by Motorola.

See also: "Working With Inline Assembly for PowerPC" on page 229.

The topics in this section are:

- Low-Level Processor Synchronization
- Floating-Point Functions
- Byte-Reversing Functions
- Setting the Floating-Point Environment
- Floating-Point Instructions for the 603 and 604
- Rotating the Contents of a Variable

Low-Level Processor Synchronization

These functions perform low-level processor synchronization.

```
void __eieio(void) /* Enforce In-Order Execution of I/O */

void __sync(void)  /* Synchronize */

void __isync(void) /* Instruction Synchronize */
```

For more information on these functions, see the instructions `eieio`, `sync`, and `isync` in *PowerPC Microprocessor Family: The Programming Environments* by Motorola.

Floating-Point Functions

These functions generate inline instructions that take the absolute value of a number.

```
int __abs(int);       /* Absolute value of an integer */

float __fabs(float);  /* Absolute value of a float */

float __fnabs(float); /* Negative absolute value of a float */

long __labs(long);    /* Absolute value of a long int */
```

Byte-Reversing Functions

These functions generate inline instructions than can dramatically speed up certain code sequences, especially byte-reversal operations

```
int __cntlzw(int);        /* Count leading zeros in a integer */

int __lhbrx(void *, int); /* Load halfword byte – reverse index */

int __lwbrx(void *, int); /* Load word byte – reverse index */

void __sthbrx(unsigned short, void *, int);
                  /* Store halfword byte – reverse index */

void __stwbrx(unsigned int, void *, int);
                  /* Store word byte – reverse indexed */
```

Setting the Floating-Point Environment

This function lets you change the PowerPC processor's Floating Point Status and Control Register (FPSCR). It sets the FPSCR to its argument and returns the original value of the FPSCR.

```
float __setflm(float);
```

This example shows how to set and restore the FPSCR:

```
double old_fpscr;
oldfpscr = __setflm(0.0); /* Clear all flag/exception/mode bits
                             and save the original settings */

/* Peform some floating point operations */

__setflm(old_fpscr);  /* Restore the FPSCR */
```

Floating-Point Instructions for the 603 and 604

These floating-point instructions, which are available only on the PowerPC 603 and 604, can speed up certain types of graphics code.

WARNING! On a computer with a PowerPC 601, they will raise an illegal instruction exception and may crash your program.

```
float __fres(float); /* Floating Reciprocal Estimate Single */

double __fsqrte(double);
               /* Floating Reciprocal Square Rooot Estimate */

double __fsel(double, double, double) /* Floating Select */
```

Rotating the Contents of a Variable

These functions rotate the contents of a variable to the left.

```
int __rlwinm(int, int, int, int);
  /* Rotate Left Word Immediate, then AND with Mask */

int __rlwnm(int, int, int, int);
  /* Rotate Left Word, then AND with Mask */

int __rlwimi(int, int, int, int, int);
  /* Rotate Left Word Immediate then Mask Insert */
```

Please note that the first argument to `__rlwimi` is overwritten.

TIP: You can use inline assembly for a complete assembly language function. See "Working With Inline Assembly for PowerPC" on page 229.

14

Target Settings for 68K

This chapter discusses each of the settings panels that affect code generation for 68K development. By modifying the settings for the individual items within a panel you control the compiler, linker, and other aspects of code generation.

Specific details about how the compiler and linker work for 68K development, such as compiler pragmas, linker symbols and so forth, is found in "C and C++ for 68K" on page 301 and "Pascal for 68K" on page 323.

The sections in this chapter are:

- Target Settings Overview
- Settings Panels for 68K

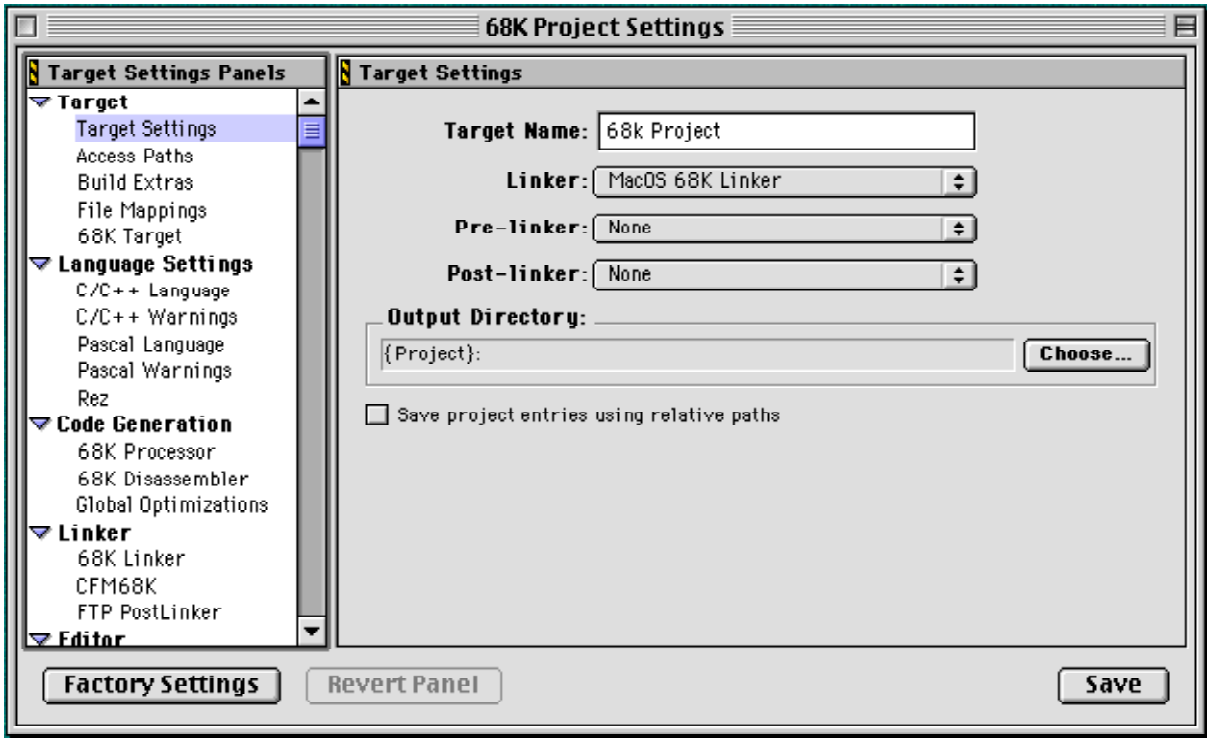
Target Settings Overview

Each target in an IDE project has its own individual settings. These settings control a variety of features such as compiler options, linker output, error and warning messages, and so forth. You modify these settings through the Target Settings dialog box. This interface is fully explained in the *IDE User Guide*.

In brief, you control compiler and linker behavior for a particular target by modifying settings in the appropriate settings panels in the Target Settings dialog box. To open any settings panel, choose ***Target*** **Settings** from the **Edit** menu, where ***Target*** is the current target in the CodeWarrior project. Or, go to the Target view of the Project window and double-click the target of interest.

When you do, the Target Settings dialog box appears, as shown in Figure 14.1.

Figure 14.1 Target Settings dialog box



Select the panel you wish to see from the hierarchical list of panels on the left side of the dialog box. When you do, that panel appears. You can then modify the settings to suit your needs.

When you modify the settings on a panel, you can restore the previous values by using the **Revert Panel** button at the bottom of the dialog box. To restore the settings to the factory defaults, use the **Factory Settings** button at the bottom of the panel.

TIP: Use project stationery when you create a new project. The stationery has all settings in all panels set to reasonable or default values. You can create your own stationery file with your preferred

settings. Modify a new project to suit your needs, then save it in the stationery folder. See the *IDE User Guide* for details.

Settings Panels for 68K

This section discusses those panels that are specific to 68K development, and the purpose and effect of each setting. The panels are:

- Target Settings
- 68K Target
- Rez
- 68K Processor
- 68K Disassembler
- 68K Linker
- CFM68K
- Debugger Settings

Settings panels of more general interest are discussed in other CodeWarrior manuals. Table 14.1 lists several panels and where you can find information about them.

Table 14.1 Where to find information on other settings panels

Panel	Manual
Access Paths	*IDE User Guide*
Build Extras	*IDE User Guide*
File Mappings	*IDE User Guide*
C/C++ Language	*C Compilers Reference*
C/C++ Warnings	*C Compilers Reference*
Pascal Language	*Pascal Reference*
Pascal Warnings	*Pascal Reference*

Panel	Manual
Global Optimizations	*IDE User Guide*
Custom Keywords	*IDE User Guide*

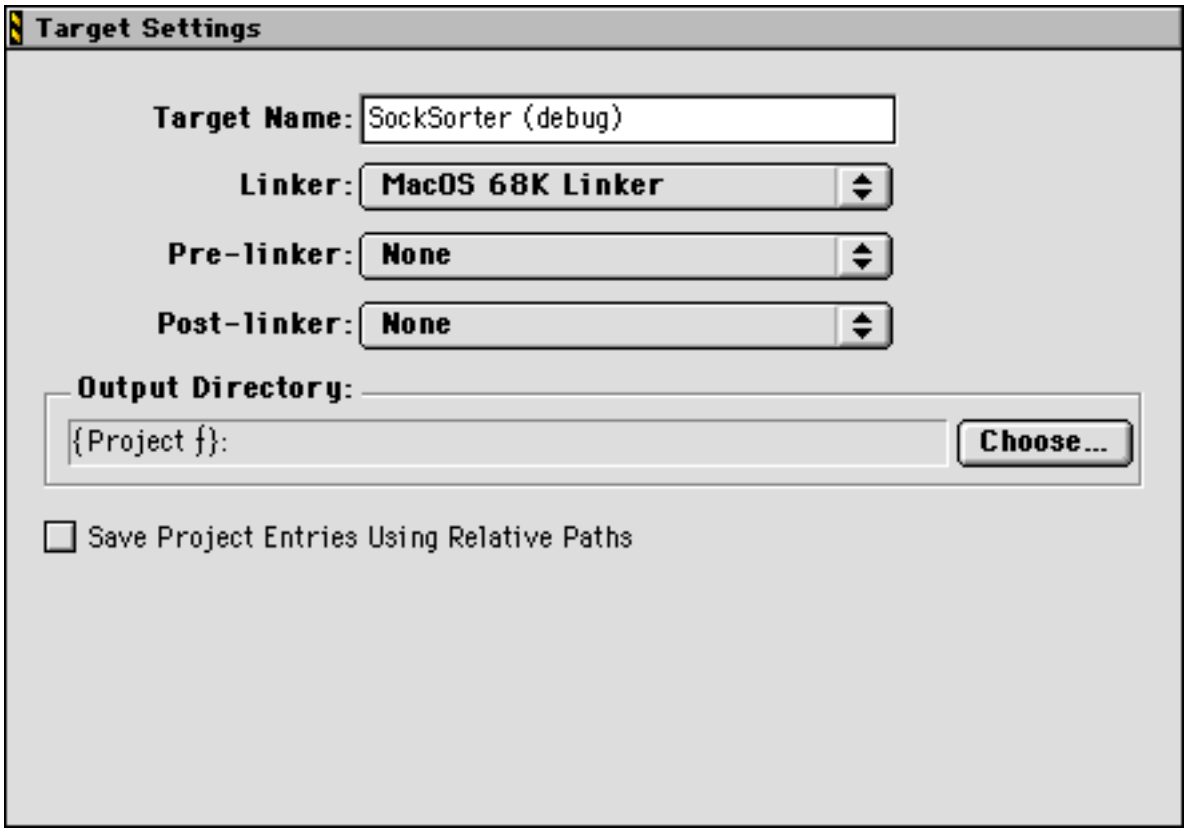
Target Settings

The Target Settings *dialog box* contains a Target Settings *panel*. The dialog box and the panel are not the same. The dialog box displays all panels, one at a time. The Target Settings *panel* is one of those panels.

The Target Settings panel, shown in Figure 14.2, is perhaps the most important panel in CodeWarrior. This is the panel where you pick your target. When you select a linker in the Target Settings panel, you specify the target operating system and/or chip. The other panels listed in the Settings dialog box will change to reflect your choice.

Because the linker choice affects the visibility of other related panels, you must set your target first before you can specify other target-specific options like compiler and linker settings.

Figure 14.2 The Target Settings panel



NOTE: The Target Settings panel is not the same as the 68K Target panel. You specify the target in the Target Settings panel. You set other target-specific options in the 68K Target panel.

The items in this panel are:

Target Name
Linker
Pre-Linker
Post-Linker
Output Directory
Save Project Entries Using Relative Paths

Target Name

Use the Target Name text field to set or change the name of a target. When you use the Targets view in the Project window, you will see the name that you have set.

The name you set here is *not* the name of your final output file. It is the name you assign to the target for your personal use. The name of the final output file is set in the 68K Linker panel.

Linker

Choose a linker from the items listed in the Linker pop-up menu. For 68K, use **Mac OS 68K Linker**.

Pre-Linker

Some targets have pre-linkers that perform work on object code before it is linked. There is no pre-linker for 68K development.

Post-Linker

Some targets have post-linkers that perform additional work (such as object code format conversion) on the final executable.

Output Directory

This is the directory where your final linked output file will be placed. The default location is the directory that contains your project file. Click the Choose button to specify another directory.

Save Project Entries Using Relative Paths

To add two or more files with the same name to a project, select this option. When this option is off, each project entry must have a unique name.

When this option is selected, the IDE includes information about the path used to access the file as well as the file name when it stores information about the file. When searching for a file, the IDE combines **Access Path** settings with the path settings it includes for each project entry.

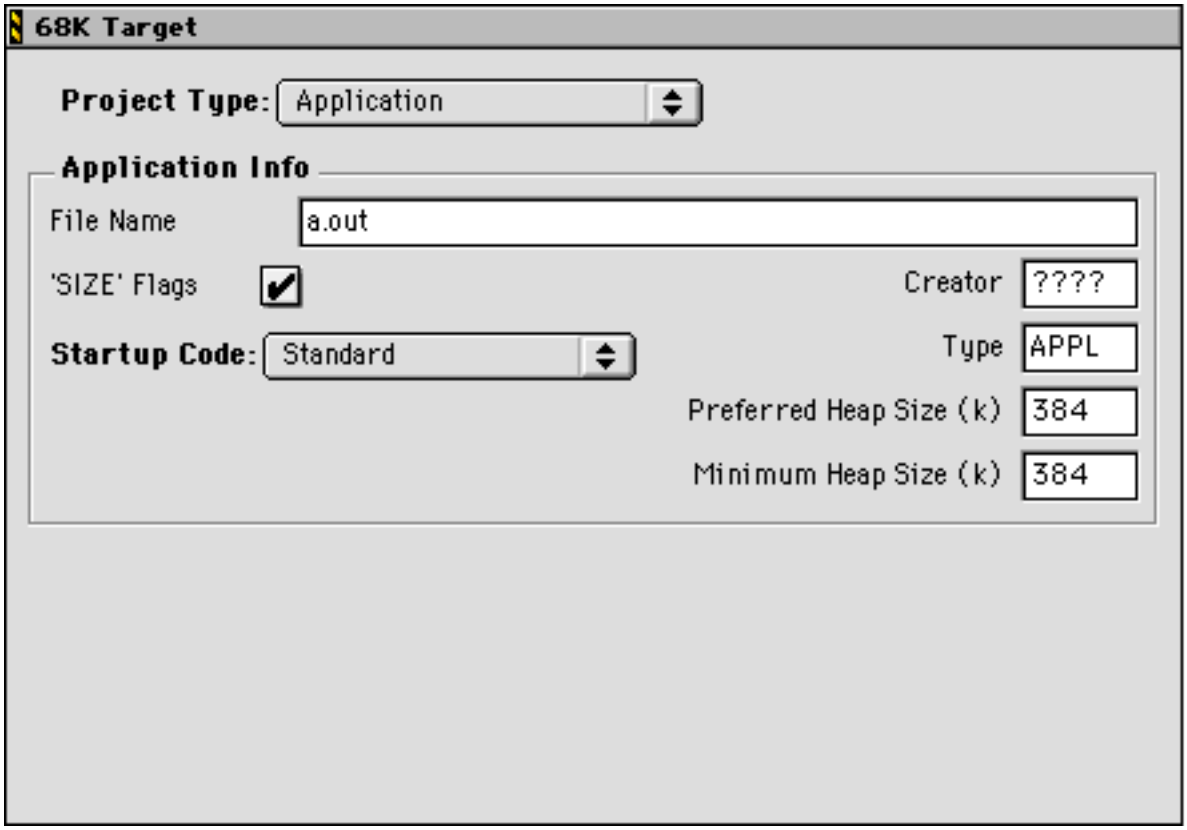
When this option is off, the IDE only records information about each project entry's file name. When searching for a file, the IDE only uses Access Paths.

68K Target

The 68K Target panel, shown in Figure 14.3, is where you set the name of your final output file.

The settings you can specify in this panel depend on the type of project you are creating.

Figure 14.3 The 68K Target panel.



The options in this panel change depending on the item chosen in the **Project Type** pop-up menu. These topics discuss the options that appear for each item you choose in the pop-up menu:

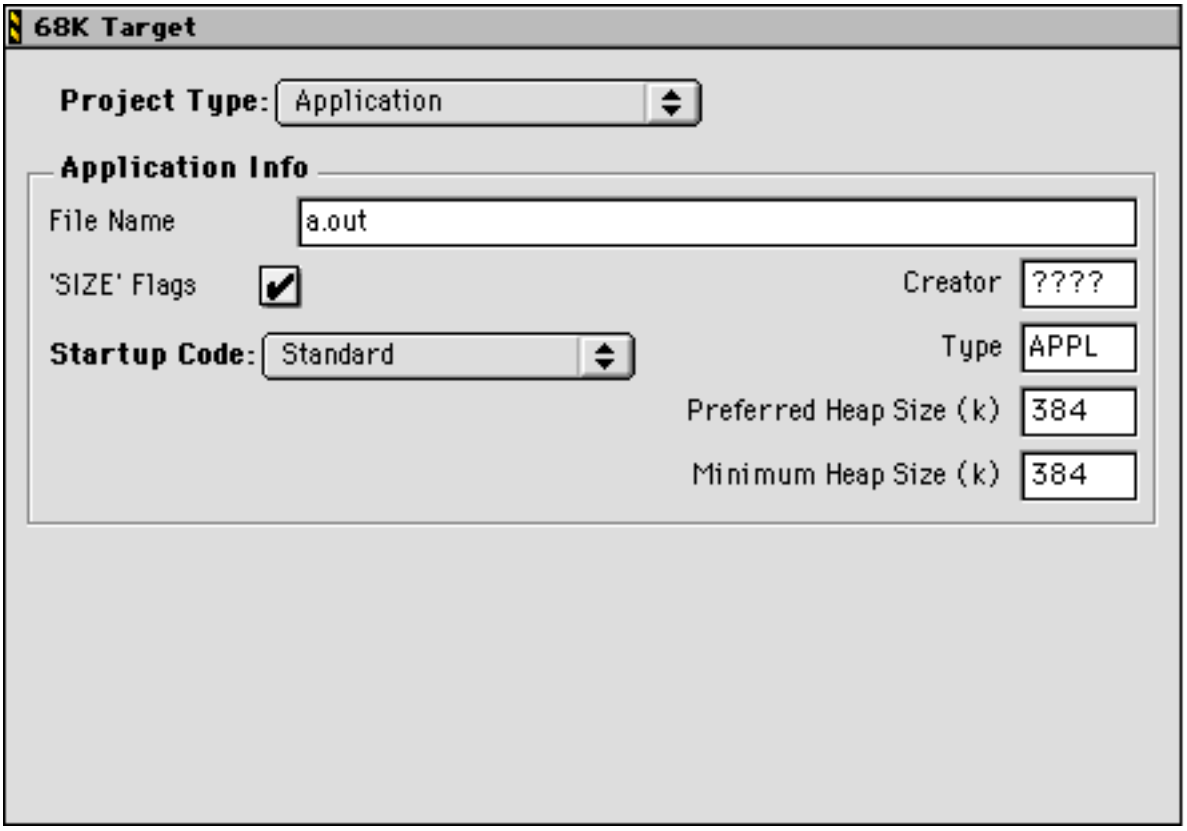
- 68K Target, Application
- 68K Target, CFM68K Application
- 68K Target, Code Resource
- 68K Target, Library
- 68K Target, CFM68K Shared Library
- 68K Target, MPW Tool

NOTE: Some items you see in the 68K Target panel are not described in this chapter because they don't apply to 68K software development.

68K Target, Application

When **Application** is chosen in the **Project Type** pop-up menu (Figure 14.4), the IDE generates a Mac OS 68K application.

Figure 14.4 68K Target Panel for a Mac OS application



The items in this panel are:

- File Name
- 'SIZE' Flags
- Startup Code
- Creator
- Type
- Preferred Heap Size
- Minimum Heap Size

For information on the other options in the 68K Target panel, see "68K Target" on page 255.

File Name

Contains the name of the application generated by the IDE.

'SIZE' Flags

Specifies the contents of the 'SIZE' generated and included in the application. See Table 14.2 for information on the items in this pop-up menu.

Table 14.2 The 'SIZE' resource options

Turn this option on…	If your application…
`acceptSuspend-ResumeEvents`	Handles suspend and resume events as well as deactivate and activate events. These events are sent when your application is brought to the foreground or sent to the background.
`canBackground`	Supports null events when it is in the background. If you turn this option off, the Finder sends your application only update events.
`doesActiveOnFGSwitch`	Does not need to receive an activate (or deactivate) event after it has come to the foreground (or background).
`onlyBackground`	Contains no user interface and is meant to run in the background.
`getFrontClicks`	Supports front clicks which bring it to the foreground. Front clicks are mouse-down and mouse-up events.
`acceptAppDiedEvents`	Needs to know whether the application which it launched was quit or crashed.
`is32BitCompatible`	Is 32-bit clean.

Turn this option on…	If your application…
`isHighLevelEventAware`	Supports AppleEvents or other high level events. If you turn this option off, the Finder does not send high level events to your application.
`localAndRemoteHLEvents`	Handles Apple Events and other high-level events from remote computers. If you turn this option off, the Finder does not send your application AppleEvents from remote computers.
`isStationeryAware`	Supports stationery documents. If you turn this option off, the Finder handles stationery documents for you.
`useTextEditServices`	Supports inline text services introduced with System 7.
`isDisplayManagerAware`	Handles the Display Notice event, which tells your application to move its windows after the monitor settings have changed. If you turn this option off, the Display Manager moves your application's windows so they don't disappear off the screen.

Startup Code

By default, CodeWarrior adds code to your program that runs at startup to initialize static variables and handle other housekeeping tasks. The **Startup Code** pop-up menu lets you choose the startup code CodeWarrior adds at link time. The **Startup Code** options are shown in Table 14.3.

Table 14.3 The Startup Code options

This option…	Adds…
Standard	Standard start-up code.
The Debugger-Aware	Startup code that lets you use Steve Jasik's The Debugger to debug multiple-segment projects.
Custom	No startup code. You must create your own.

Creator

The Finder identifies your application with its **Creator** code, a unique four-character sequence. Enter the code without single quotes. For example, type `SSrt` instead of `'SSrt'`.

Your application's Creator code must not conflict with any other application's.

NOTE: To be absolutely sure your Creator code is unique, register it with Apple Computer, Inc., at Macintosh Developer Technical Support. You should be able to find out more information about this topic on the World-Wide Web at `http://devworld.apple.com`

Type

A Mac OS application's Type code must always be `'APPL'`, or `'appe'`. The default value for this field is `'APPL'`. If you use `'appe'`, you are writing a background application. Since there are some restrictions for what background applications can do, you should refer to other sources of information for applications of type 'appe'. One such source would be Apple Technical Notes, accessible on the World-Wide Web at `http://devworld.apple.com`

Other types of projects have different Type codes. For example, a code resource might have the Type 'INIT' or 'PRER', and a shared library might have the Type 'shlb'.

Enter the code without single quotes. For example, type APPL instead of 'APPL'.

Preferred Heap Size

The recommended amount of RAM your Mac OS application wants, in KBytes. In the Finder's Get Info dialog, this is the number in the Suggested Size field and the default value for the Preferred Size field.

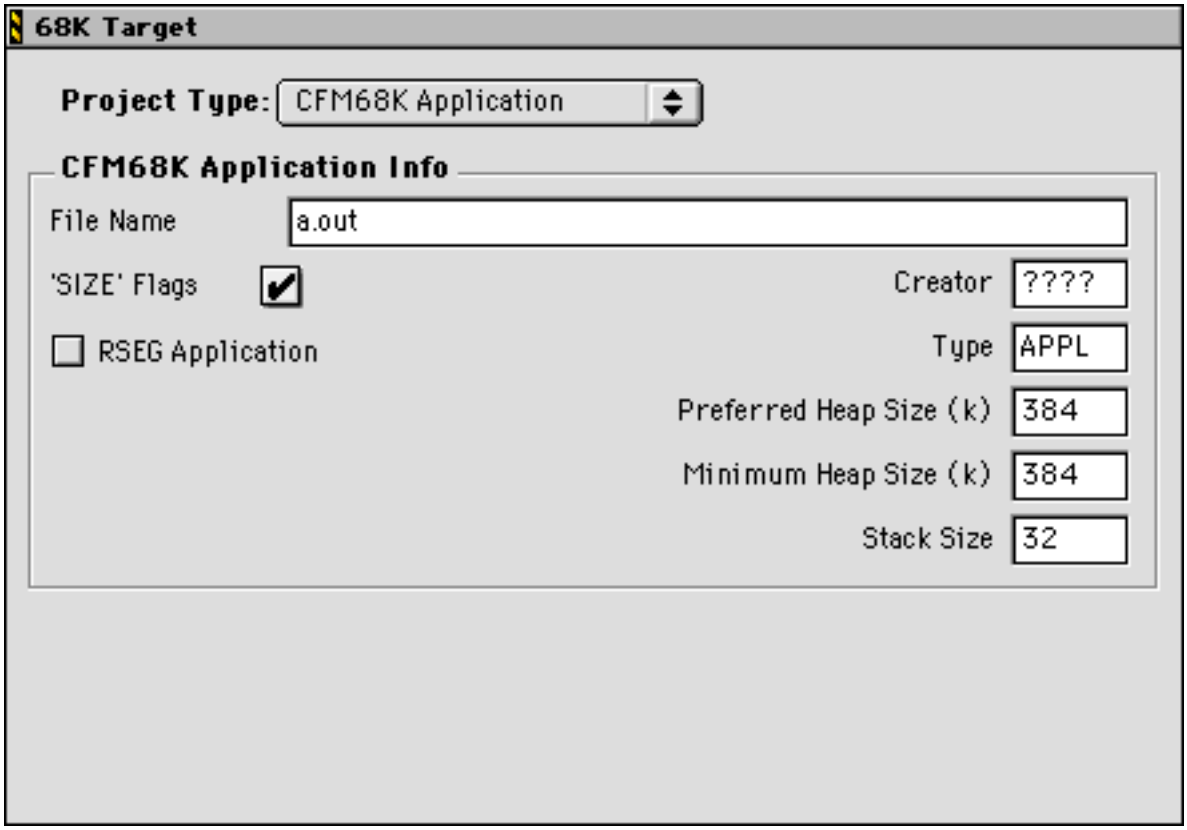
Minimum Heap Size

The minimum amount of RAM your application needs, in KBytes. In the Finder's Get Info dialog, this is the default value for the Minimum Size field.

68K Target, CFM68K Application

When **CFM68K Application** is chosen in the **Project Type** pop-up menu (Figure 14.5), the IDE generates a 68K CFM68K application.

Figure 14.5 68K Target Panel for a 68K CFM68K application



Choosing this item in the **Project Type** pop-up menu shows the same options shown when 68K Target, Application is chosen. Additional items in this panel are:

RSEG Application Stack Size

Some 68K applications use the Code Fragment Manager for 68K. This software is contained in a software extension called `CFM-68K Runtime Enabler`, that only works on 68K computer with MC68020 or better chip. It contains its code in a single `'RSEG'` resource. It's useful if you want the advantages of shared libraries on a 68K 68K computer. You must have the `CFM-68K Runtime Enabler` installed on the 68K computer that you want to run your application on.

For information on the other options in the 68K Target panel, see "68K Target" on page 255.

RSEG Application

Select this item to place the PEF container in an `'rseg'` code resource in the application file instead of the file's data fork.

Stack Size

An application's stack contains local variables, return addresses, and temporary information. If your application uses a lot of stack space (for example, it makes extensive use of recursive functions, or uses large arrays local to a routine), you may need to change the default size of the stack.

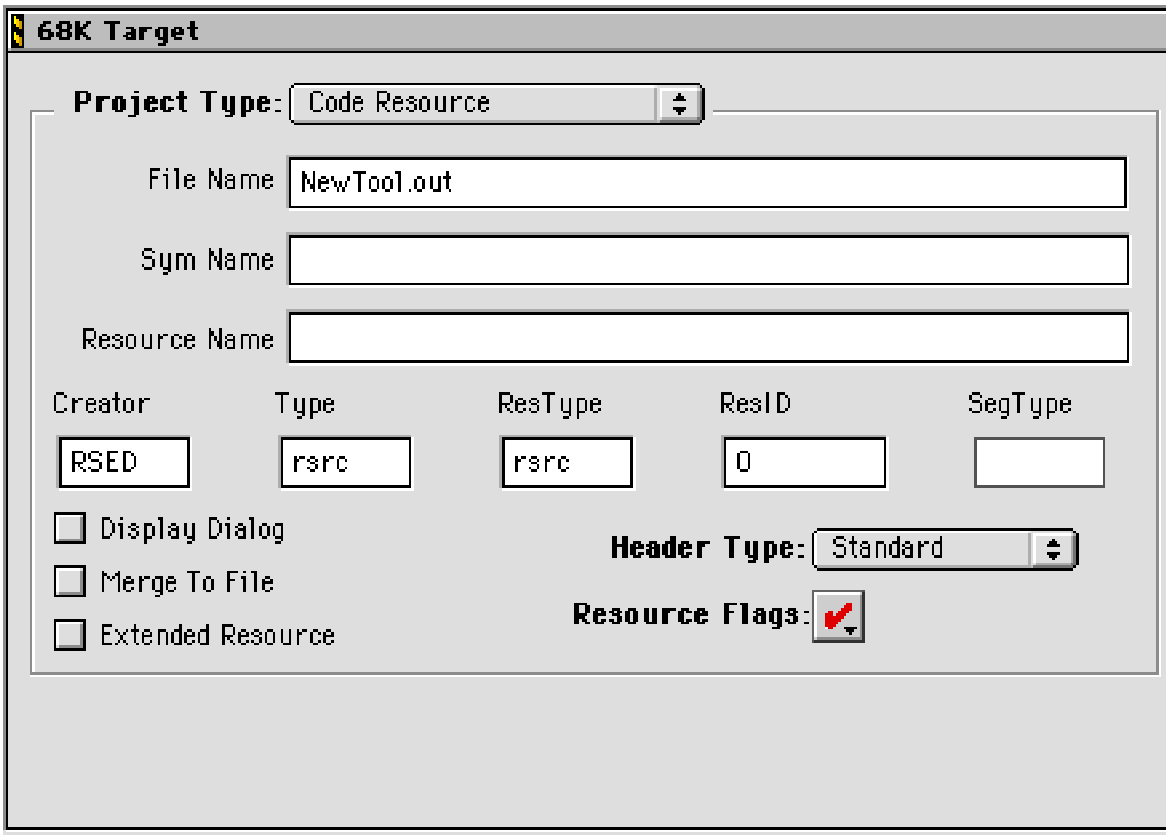
In a PowerPC or CFM68K 68K application, you can set the size in the Stack Size field. The default size is 64K.

In a 68K 68K application, you must set the stack size within your application's code by calling the function `SetApplLimit()`. For more information, see Chapter 1, "Introduction to Memory Management," in *Inside Macintosh: Memory*.

68K Target, Code Resource

When **Code Resource** is chosen in the **Project Type** pop-up menu (Figure 14.6), the IDE generates a 68K executable code resource.

Figure 14.6 68K Target Panel for a 68K code resource



The items in this panel are:

- File Name
- Sym Name
- Resource Name
- Creator
- Type
- ResType
- ResID
- SegType
- Display Dialog
- Extended Resource
- Merge to File
- Resource Flags
- Header Type

By default, the CodeWarrior Linker saves your resource file in the same folder as your project, with the name specified in the File

Name field. You save the file somewhere else with the options named **Display Dialog** and **Merge To File**. These options are discussed in Table 14.4.

Table 14.4 Choosing where to save your resource

To save the resource here…	Do this…
In a new file in the same folder as your project	Turn off Display Dialog, Turn off Merge To File, and Enter the name in the File Name field.
In an existing file in the same folder as your project	Turn off Display Dialog, Turn on Merge To File, and Enter the name in the File Name field.
In a new file in a folder of your choosing	Turn on Display Dialog and Turn off Merge To File.
In an existing file in a folder of your choosing	Turn on Display Dialog and Turn on Merge To File.

For information on the other options in the 68K Target panel, see "68K Target" on page 255.

File Name

Specifies the name of the file in which the resource will be placed. This field will be used when the **Display Dialog** option is de-selected. If **Merge To File** is selected, the code resource will be added to file with this name in the same folder as the project. If **Merge To File** is deselected, the IDE will delete the contents (or create) the file with this name in the same folder as the project before adding the code resource.

Sym Name

Specifies the name of the file in which the code resource's SYM information (debugging information) will be placed.

Resource Name

Specifies the name of the code resource's name in the resource fork.

NOTE: If you're creating a multi-segment resource, the linker only names the main resource, not its segments.

Creator

Specifies which application opens your resource file. By default, this field contains 'RSED', which is the Creator for the ResEdit visual resource editor application. Enter the code without single quotes. For example, type RSED instead of 'RSED'.

Type

Specifies the type of the file that contains the code resource. For example, 'INIT' is used as the Type for a system extension and 'PRER' is used for a Chooser extension. Enter the code without single quotes. For example, type INIT instead of 'INIT'.

To find out the proper values for your resource, check the documentation that describes how to create it. For example, if you are creating a Photoshop plugin, check the Adobe documentation that tells how to write Photoshop plugins.

ResType

Identifies what kind of resource you're creating. For example, 'XCMD' is for a HyperCard XCMD, 'XDEF' is for a window definition, and 'MDEF' is for a menu definition. Enter the code without single quotes. For example, type XDEF instead of 'XDEF'.

For 68K multi-segment code resources, this field specifies the resource type of the main segment. Subsequent resources are given the type specified by **SegType**.

ResID

The ID number, along with the resource type, uniquely identifies the resource to the system software. In general, the number may be between 128 and 32,767. 68K reserves ID numbers from -32768 to 127 for most resource types. Consult *Inside Macintosh:Toolbox Essentials* for more information.

SegType

Specifies the resource type for all the resource segments besides the main segment in a 68K multi-segment resource. The resource type for the main segment is the **ResType**.

Display Dialog

Shows a dialog box prompting you for a file to store the code resource in. If this item is selected, the file entered in File Name is ignored. If **Merge To File** is selected, the code resource will be added to file you've selected. If **Merge To File** is deselected, the IDE will delete the contents the file (or creates the file if it doesn't already exist) you've selected before adding the code resource.

Extended Resource

If your code resource is larger than 32K bytes, select this option. Used with the **!!!Large selected in Code Model!!!** option, this item ensures that the code resource uses the proper addressing mode in its instructions when accessing data and instructions.

Merge to File

Specifies that the code resource should be added to the file specified by File Name or selected in the file dialog box when Display Dialog is selected. If selected, the code resource is added to the file. If deselected, the IDE deletes the file's contents (or creates the file if it doesn't already exist) before adding the code resource.

Resource Flags

Specifies the Resource Manager flags to set for the resource. Table 14.5 describes what each resource attribute flag does. For more in-

formation on these flags, refer to the chapter called "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox,* published by Addison-Wesley.

Table 14.5 Resource attributes

Turn this attribute on…	To…
Preload	Load the resource into memory as soon as the Resource Manager opens it. When the application is launched, anything set with preloaded is loaded.
Protected	Protect the contents, name, and ID of the resource. If a resource is protected, it cannot be removed from the file to which it belongs.
Locked	Lock the resource in the heap so the system cannot move it. It gets locked into the heap's low memory address space.
Purgeable	Purge the resource from memory when the system needs to make more memory available.
System Heap	Place the resource in the system heap. If your resource is too large, it is placed in the application heap. If you turn this option off, your resource is placed in the application heap.

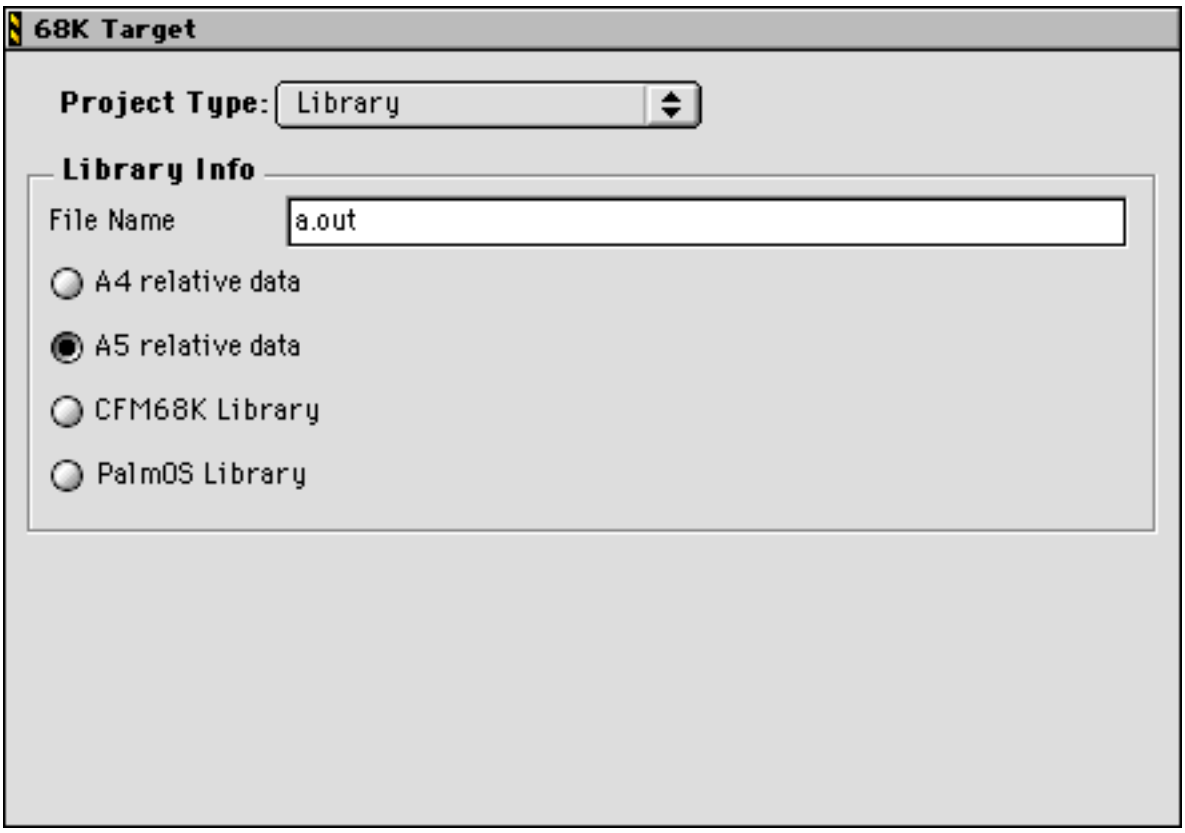
Header Type

Specifies the type of code resource header to place at the beginning of the code resource.

68K Target, Library

When **Library** is chosen in the **Project Type** pop-up menu (Figure 14.7), the IDE generates a 68K static library.

Figure 14.7 68K Target Panel for a static library



The items in this panel are:

File Name

A4 relative data

A5 relative data

CFM68K library

PalmOS library

For information on the other options in the 68K Target panel, see "68K Target" on page 255.

File Name

Specifies the name of the file in which the library is stored.

A4 relative data

Specifies that the library's instructions should use the 68K processor's A4 register to refer to global data. Libraries used to create Mac OS code resources should make sure this option is selected.

A5 relative data

Specifies that the library's instructions should use the 68K processor's A5 register to refer to global data. Libraries used to create Mac OS applications should make sure this option is selected.

CFM68K library

Specifies that the IDE should generate a Mac OS CFM68K library.

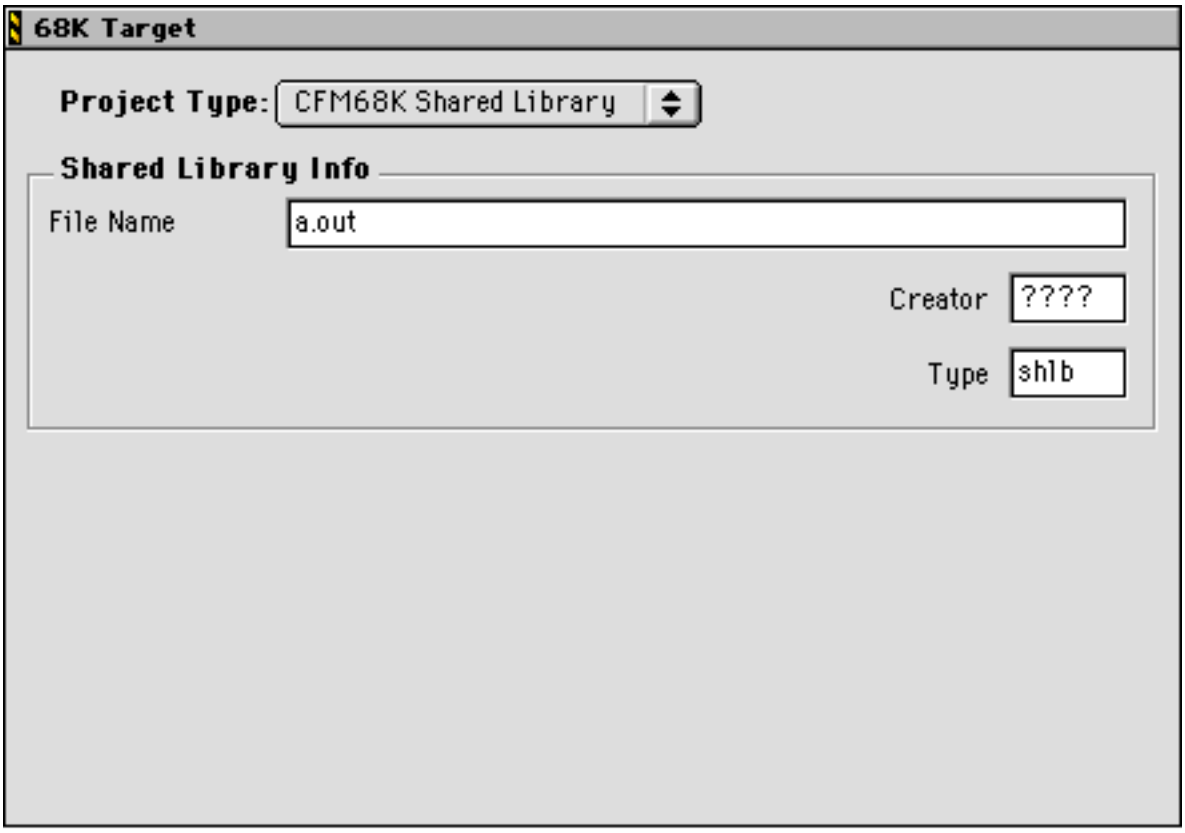
PalmOS library

Specifies that the IDE should generate a library for Palm OS.

68K Target, CFM68K Shared Library

When **CFM68K Shared Library** is chosen in the **Project Type** pop-up menu (Figure 14.8), the IDE generates a CFM68K library.

Figure 14.8 68K Target Panel for a Mac OS CFM68K static library



Items in this panel are:

[File Name](#) [Creator](#)

[Type](#)

For information on the other options in the 68K Target panel, see "68K Target" on page 255.

File Name

Specifies the name of the file in which the library is stored.

Creator

Specifies which application opens your resource file. By default, this field contains `'????'`. Enter the code without single quotes. For example, type ???? instead of `'????'`.

Type

Specifies the type of the file that contains the code resource. By default, the CodeWarrior IDE sets this field to 'shlb', which is for a shared library. If you want the system software to treat your library like an application, set this field to 'APPL'`instead`. Enter the code without single quotes. For example, type `shlb` instead of `'shlb'`.

68K Target, MPW Tool

Choosing **MPW Tool** in the **Project Type** pop-up menu, the IDE generates an application that runs under Apple's Macintosh Programmer's Workshop (MPW) development environment. Such an application, called a *tool* in MPW, will not run when launched in Mac OS.

Items in this panel are:

File Name

File Name

Specifies the name of the file in which the tool is stored.

Figure 14.9 68K Target Panel for an MPW tool

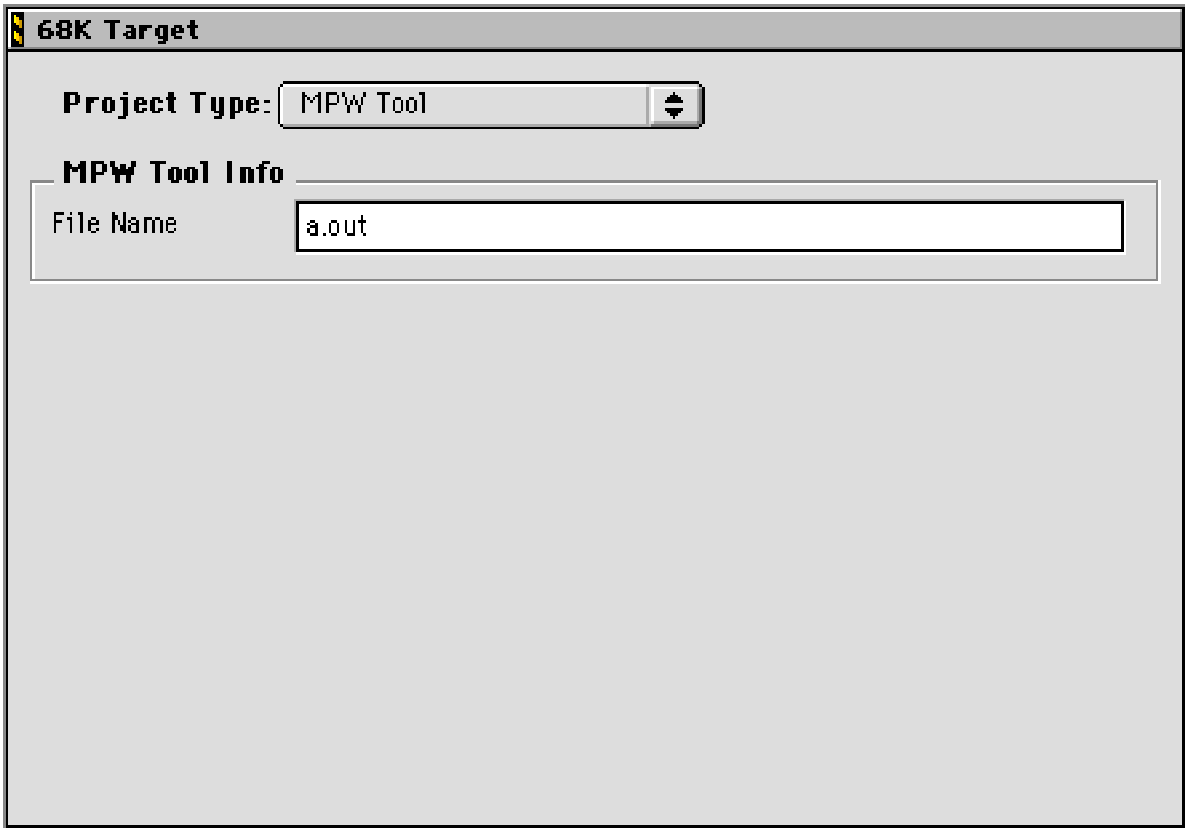
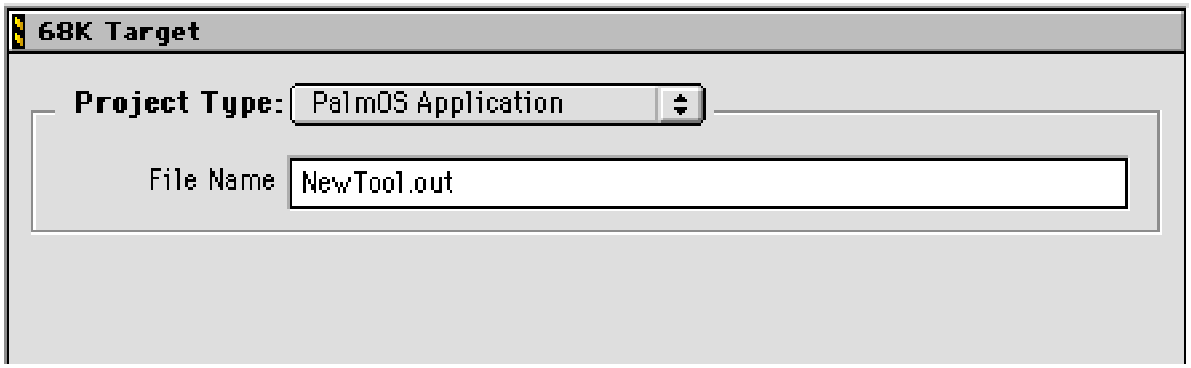


Figure 14.10 68K Target Panel for PalmOS Application



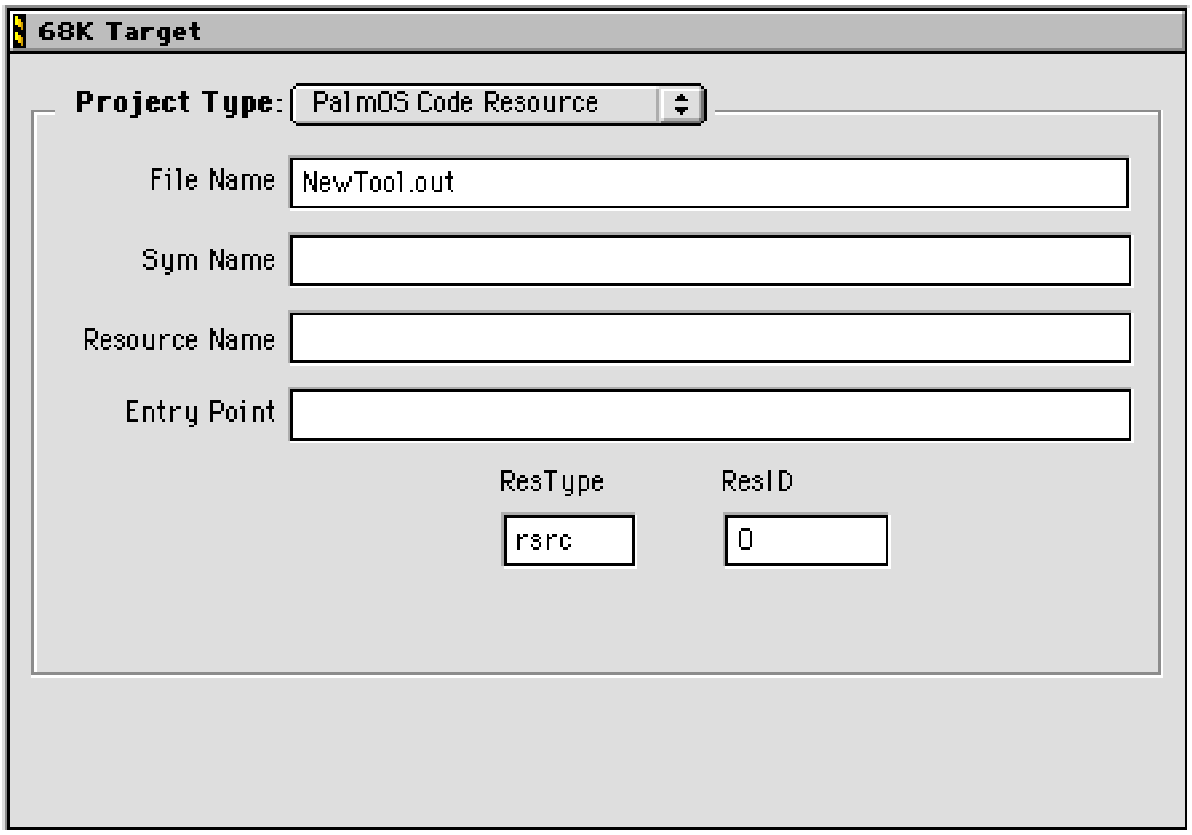
Items in this panel are:

File Name

File Name

Specifies the name of the file in which the tool is stored.

Figure 14.11 68K Target Panel for PalmOS Code Resource



Items in this panel are:

File Name — Sym Name

Resource Name

ResType — ResID

File Name

Specifies the name of the file in which the tool is stored.

with this name in the same folder as the project before adding the code resource.

Sym Name

Specifies the name of the file in which the code resource's SYM information (debugging information) will be placed.

Resource Name

Specifies the name of the code resource's name in the resource fork.

ResType

Identifies what kind of resource you're creating. For example, `'XCMD'` is for a HyperCard XCMD, `'XDEF'` is for a window definition, and `'MDEF'` is for a menu definition. Enter the code without single quotes. For example, type `XDEF` instead of `'XDEF'`.

For 68K multi-segment code resources, this field specifies the resource type of the main segment. Subsequent resources are given the type specified by **SegType**.

ResID

The ID number, along with the resource type, uniquely identifies the resource to the system software. In general, the number may be between 128 and 32,767. 68K reserves ID numbers from -32768 to 127 for most resource types. Consult *Inside Macintosh:Toolbox Essentials* for more information.

Rez

The **Rez** settings panel, shown in Figure 14.12, allows you to control how rez compiles a source file, and how derez decompiles a resource file.

The settings are:

Escape Control Chars	Max String Width
Filter Mode	Suppress Warnings for Redeclared Resource Types
Alignment	Script Language
Prefix File	

Escape Control Chars

When selected, the **Escape Control Chars** checkbox allows control characters to be printed.

Max String Width

The **Max String Width** edit field allows you to specify the maximum string width of the derez output. You must specify a width between two and 120. If you enter a number outside this range, derez will give you a warning and use the default value of 80.

Filter Mode

The **Filter Mode** edit field allows you to specify the resource types to be derezed or the resource types not to be derezed, depending on which option is selected from the Filter Mode pop-up menu. You may enter any four-character type in this field. Types must be surrounded by single quotes (') and separated by commas (,). There are two options in the pop-up menu:

- **Skip these resource types** — if selected, the types entered in the edit field will not be derezed; every other type will be derezed.
- **Only derez the resource types** — if selected, the types specified in the edit field will be the only types to be derezed; all others will be skipped.

Suppress Warnings for Redeclared Resource Types

When selected, the **Suppress Warnings for Redeclared Resource Types** checkbox will not allow warnings to be shown for redeclared resource types.

Alignment

The **Alignment** pop-up menu is currently not supported.

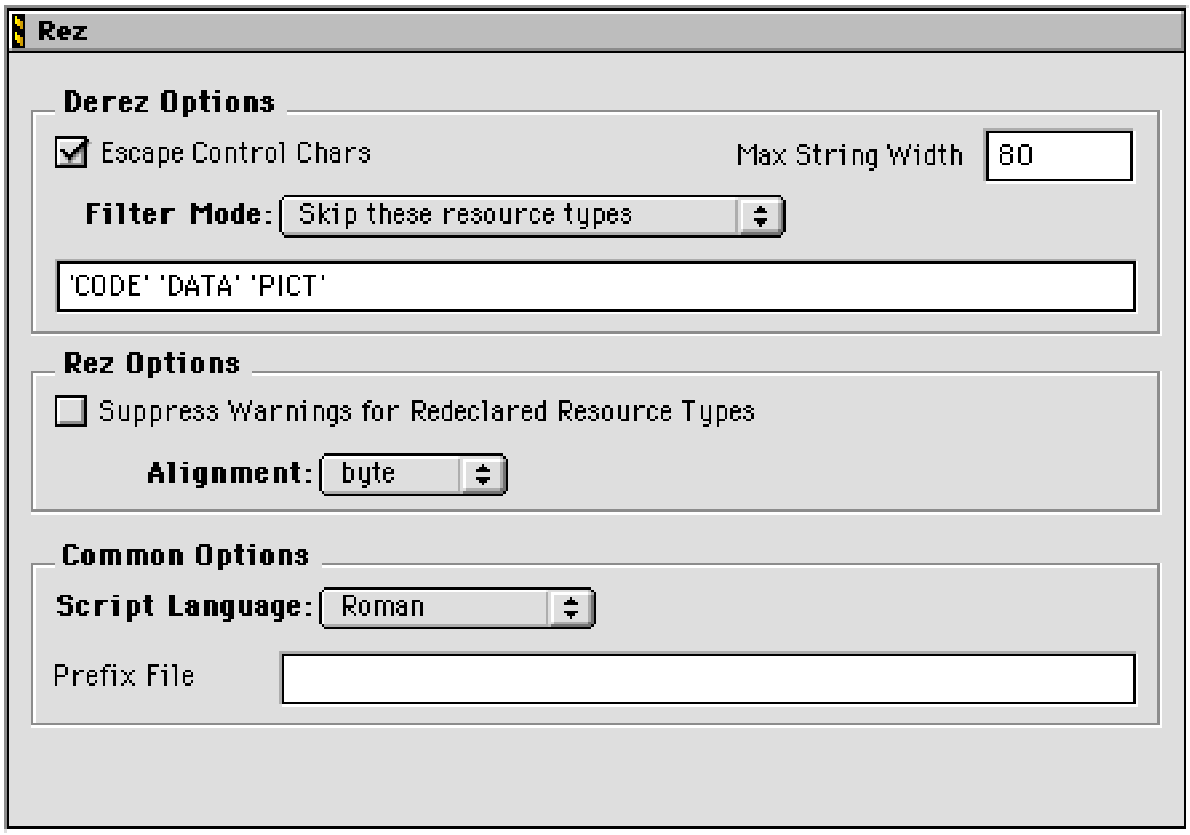
Script Language

The **Script Language** pop-up controls which script system Rez uses when it compiles multi-byte characters. For example, if you have the Japanese Language Kit installed and your strings contain Kanji characters, then you would need to change the script language from **Roman** to **Japanese**.

Prefix File

The **Prefix File** edit field allows you to specify the file to be read in by the compiler before it processes the source file or the resource file in the case of derez. The prefix file is where derez gets the resource type definitions that it uses to decompile the resources.

Figure 14.12 The Rez settings panel.



68K Processor

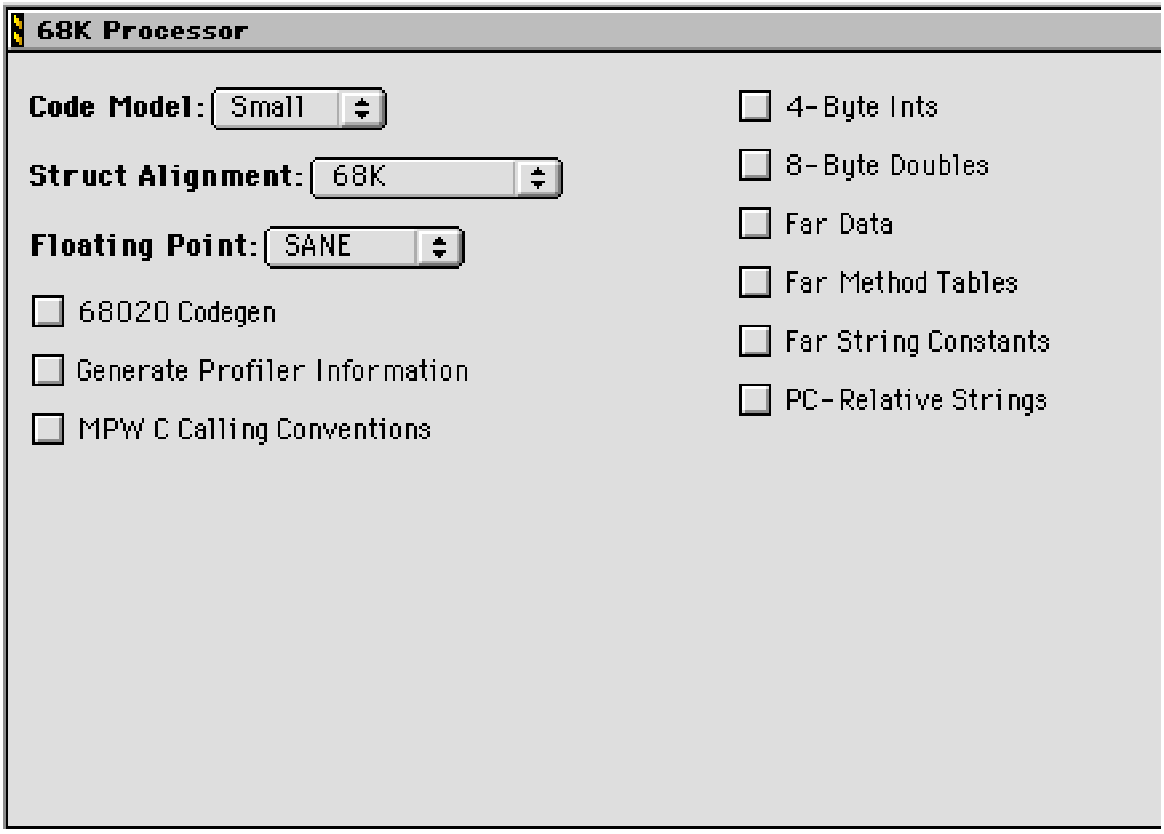
The 68K Processor panel, shown in Figure 14.13, is where you control settings related to code generation for the 68K platform.

The items in this panel are:

- Code Model
- Struct Alignment
- Floating Point
- 68020 Codegen
- 4-Byte Ints
- 8-Byte Doubles
- Far Data
- Far Method Tables

Generate Profiler Information Far String Constants
MPW C Calling Conventions PC-Relative Strings

Figure 14.13 The 68K Processor panel.



Code Model

Selects how data and instructions are accessed in the object code generated by the compiler. Table 14.6 describes the items in this pop-up menu.

Table 14.6 Code Model options

If you choose...	the code uses this addressing
Smart Model	When one function calls another function and the linker is sure they're in the same segment, this model uses relative addressing (a 16-bit address). Otherwise, it uses absolute addressing (a 32-bit address).
Small Model	This model uses relative addressing (a 16-bit address) for all function calls. Small Model is best for small code resources and applications (under 32K), or for code resources and applications in which all segments are under 32K.
Large Model	This model uses absolute addressing (a 32-bit address) for all function calls. Large Model is useful when you have a source file that generates more than 32K of code, or when the linker generates an out-of-range link error.

This option corresponds to `#pragma far_code`, `#pragma near_code`, and `#pragma smart_code`. For more information on `#pragma statements`, refer to the *C Compilers Reference*.

Struct Alignment

Specifies how records and structures are arranged in memory. Table 14.7 describes the options in this pop-up menu.

Table 14.7 Structure alignment options

If you choose…	The linker aligns every field on…
68K	2-byte boundaries, unless a field is only 1-byte long. This is the standard alignment for 68K Macintosh computers.
68K 4-byte	4-byte boundaries.
PowerPC	Its natural boundary. For example, it aligns a 1-byte character on a 1-byte boundary and a 16-bit integer on a 2-byte boundary. The compiler applies this alignment recursively to structured data and arrays containing structured data. So, for example, it aligns an array of structured types containing an 8-byte floating point member on an 8-byte boundary. This is the standard alignment for Power Macintosh computers.

This option corresponds to `#pragma options align`.

Floating Point

Specifies which floating point library to support and what kind of instructions to generate when compiling floating point expressions.

Table 14.8 describes the items in this pop-up menu.

Table 14.8 Floating Point options

If you choose...	the compiler generates these instructions
SANE	Floating point expressions are translated to refer to Apple's SANE (Standard Apple Numeric Environment).
68881	Floating point expressions are translated into instructions for the Motorola 68881 math co-processor.
Library	Floating point expressions are translated to refer to routines in the floating point library, `fp68k.lib`.
PalmOS	Floating point expressions are generated to use the floating point routines in version 2 of Palm OS.

68020 Codegen

Specifies the use of instructions available only on MC68020, MC68030, and MC68040 processors, including integer multiplication, integer division, and bit-field operations. When selected these extra instructions are generated by the compiler. When deselected, the instructions generated by the compiler compatible with MC68000, MC68020, MC68030, and MC68040 processors.

WARNING! Before your program runs code optimized for the MC68020, use the `Gestalt()` function to make sure it is available. For more information on `Gestalt()`, see Chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

The **68020 Codegen** option corresponds to `#pragma code68020`.

Generate Profiler Information

To set up a project for the Profiler, turn on this option. When on special object code is inserted into your application that collects timing information for the profiler.

When you're ready to create the finished version of your application, turn this option off to eliminate the profiler code. When you use the IDE debugger, turn this option off since the profiler code conflicts with the debugger.

These options correspond to `#pragma profile`.

Refer to the *CodeWarrior Profiler Manual* for information on using the profiler.

MPW C Calling Conventions

Specifies that object code should conform to MPW (Macintosh Programmer's Workshop) C calling conventions. When you turn on the MPW C Calling Convention option, the compiler does the following in order to ensure compatibility with MPW C's calling conventions:

- Passes any integral argument that is smaller than 2 bytes as a sign-extended `long int`. For example, the compiler converts this declaration:

  ```
  int MPWfunc( char a, short b, int c,
               long d, char *e );
  ```

  To this:

  ```
  long MPWfunc( long a, long b, long c,
                long d, char *e );
  ```

- Passes any floating-point arguments as a `long double`. For example, the compiler converts this declaration:

  ```
  void MPWfunc( float a, double b,
                long double c );
  ```

  To this:

  ```
  void MPWfunc( long double a, long double b,
                long double c );
  ```

- Returns any pointer value in `D0` (even if the `#pragma pointers_in_D0` is off).
- Returns any 1-byte, 2-byte, or 4-byte structure in `D0`.
- If the 68881 Codegen option is on, returns any floating-point value in `FP0`.

NOTE: The MPW C Calling Conventions option is available only with the 68K compilers.

The MPW C Calling Conventions option corresponds to `#pragma mpwc`.

4-Byte Ints

Specifies the size of the C/C++ `int` data type. When selected, values of type int are 4 bytes (32 bits). When deselected, values of type `int` are 2 bytes (16 bits).

In general, turn this option on since it's easier to port your code to the Power Macintosh compiler, which always uses 4-byte `int`s. However, 2-byte `int`s are slightly more efficient on the 68K, so you may want to turn this option off when efficiency is more important.

NOTE: The 68K Pascal compiler ignores this option's setting, and an `integer` is always 2-bytes. The **PPC Processor** panel does not contain this option. In Power Macintosh C/C++, an `int` is always 4-bytes, and in Power Macintosh Pascal, an `integer` is always 2-bytes.

The 4-Byte Ints option corresponds to `#pragma fourbyteints`.

8-Byte Doubles

Specifies the size of floating point numbers. When selected, the C/C++ double data type is 8 bytes (64 bits). When deselected, the C/C++ double data type is 10 bytes (80 bits) or 12 bytes (96 bits), de-

pending on the settings of other options. See Table 14.9 and Table 14.10 for the size of floating point data types.

You can choose the size of a floating-point numbers with the 8-Byte Doubles option. In 68K C/C++, this affects variables declared as type `double`. In 68K Pascal, this affects variables declared `longreal`. In general, turn this option off since 8-byte (or 64-bit) floating-point numbers are less efficient than others. However, if you are porting code that relies on 8-byte floating-point numbers, turn this option on.

You can also choose to create code that is optimized for machines with a 68040 processor or a 68881 floating point unit. If you turn on the 68881 Codegen option in the **68K Processor** panel, the compiler uses floating-point operations and types that are designed specifically for those chips. If you create code with the 68881 Codegen option on and try to run it on a machine that does not have a 68040 or 68881, the code will crash. Turn on the 68881 Codegen option only if the code contains lots of floating-point operations, must be as fast as possible, and you're sure the code will be used only on machines that contain a 68040 or 68881.

Table 14.9 describes how to choose the size of 68K C/C++ floating-point types with the options 8-Byte Doubles and 68881 Codegen.

Table 14.9 68K C and C++ Floating Point Types

For this type…	If…	Its size is	and its range is
`double`	8-Byte Doubles is on	64 bits	`2.22507e-308` to `1.79769e+308`
	8-Byte Doubles is off and 68881 Codegen is off	80 bits	`1.68105e-4932` to `1.18973e+4932`

For this type…	If…	Its size is	and its range is
	8-Byte Doubles is off and 68881 Codegen is on	96 bits	`1.68105e-4932` to `1.18973e+4932`
`long double`	68881 Codegen is off	80 bits	`1.68105e-4932` to `1.18973e+4932`
	68881 Codegen is on	96 bits	`1.68105e-4932` to `1.18973e+4932`

Table 14.10 describes how to choose the size of the 68K Pascal floating-point type `longreal` with the options 8-Byte Doubles and 68881 Codegen.

Table 14.10 68K Pascal Floating Point Types

For this type…	If the following is true…	Its size is	and its range is
`longreal`	8-Byte Doubles is on	64 bits	`2.22507e-308` to `1.79769e+308`
	8-Byte Doubles is off and 68881 Codegen is off	80 bits	`1.68105e-4932` to `1.18973e+4932`
	8-Byte Doubles is off and 68881 Codegen is on	96 bits	`1.68105e-4932` to `1.18973e+4932`

The 8-Byte Doubles option corresponds to `#pragma IEEEdoubles`.

Far Data

Specifies how global data is stored and referred to. When on, global data is stored in far data space and is only limited in size to avail-

able memory. When off, global data is stored in near data space which is limited to 64K bytes.

This Far Data option corresponds to `#pragma far_data`.

Far Method Tables

Specifies how C++ virtual function dispatch tables (vTables) or Pascal method tables are stored and referred to. When on, method tables are stored in far data space which is limited in size to available memory. When off, method tables are stored in near data space which is limited to 64K bytes.

If you use virtual functions in a C++ project, the linker needs to store virtual function dispatch tables (or vTables) with the global data. When your application is running and calls a virtual function, it uses these tables to find the address of appropriate implementation of that virtual function.

This Far Method Tables option corresponds to `#pragma far_vtables`.

Far String Constants

Specifies how character string constants are stored and referred to. If the Far String Constants option is on, the linker stores all string constants with the rest of the global data in near space. If this option is off, the linker stores all string constants in far space

The Far String Constants option corresponds to `#pragma far_strings`. The factory setting for this option is off.

PC-Relative Strings

Stores the string constants used in a local scope in the code segment and addresses these strings with PC-relative instructions. If this option is off, the linker stores all string constants in the global data segment. Regardless of how this option is set, the compiler stores string constants used in the global scope in the global data segment. For example:

```
int x = f("Hello");// "Hello" is in the
                   // global data segment
int bar()
{
  return f("World"); //"World" is in the
}                    // local code segment
```

NOTE: Strings in C++ initialization code are always allocated in the global data segment.

If your program uses a large number of string constants, they could overflow the 64K usually reserved for global data. Turn on this option to save space in the global data space.

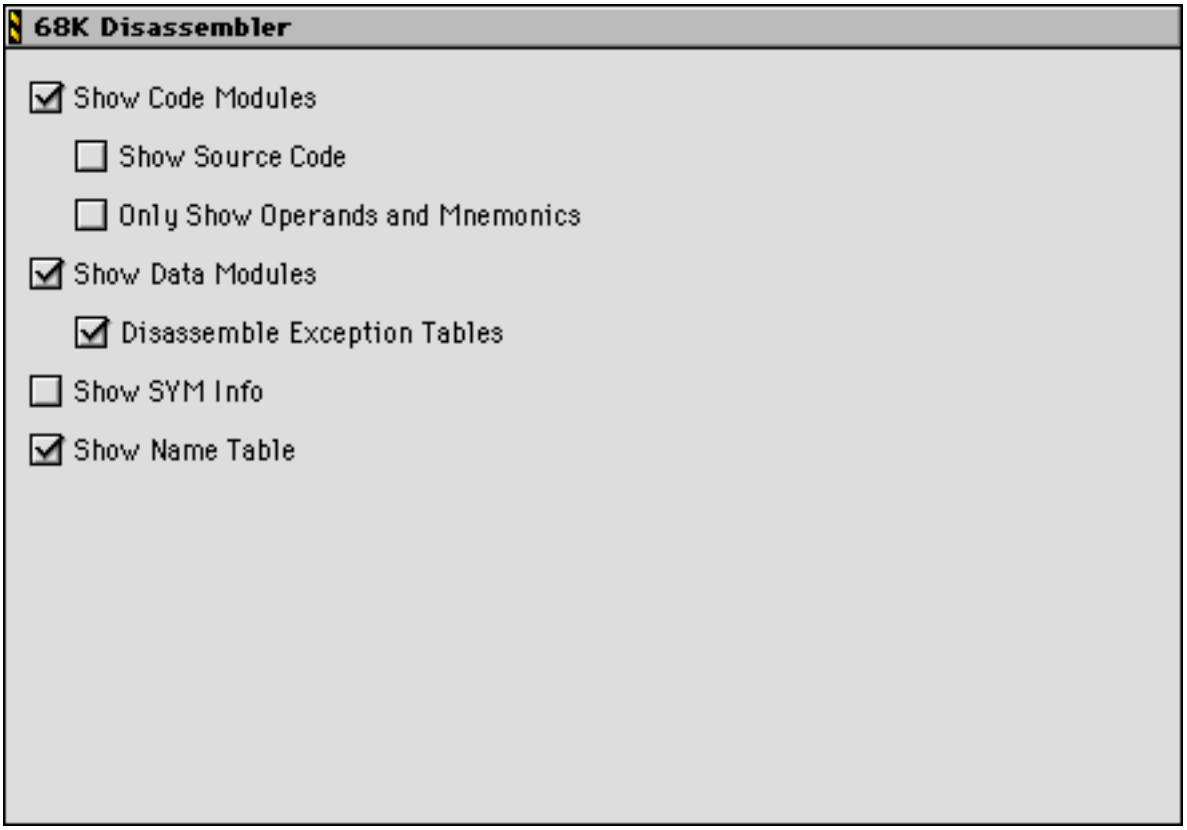
The PC-Relative Strings option corresponds to `#pragma pcrelstrings`.

68K Disassembler

The 68K Disassembler panel, shown in Figure 14.14, is where you control the information displayed when you choose the **Disassemble** command from the IDE's **Project** menu.

See the "Compiling and Linking" chapter of the *IDE User Guide* for general information about the Disassemble command.

Figure 14.14 The 68K Disassembler panel.



The items in this panel are:

Show Code Modules — Show Source Code

Only Show Operands and Mnemonics — Show Data Modules

Disassemble Exception Tables — Show SYM Info

Show Name Table

Show Code Modules

Specifies whether or not to show object code. When selected, disassemblies list object code. When deselected, disassemblies don't show object code.

Show Source Code

Specifies whether or not to show corresponding source code. When selected and SYM information is being generated, source code corresponding to the disassembled object code is also listed. When deselected, no source code is shown.

Only Show Operands and Mnemonics

Specifies whether or not to show corresponding hexadecimal values along with operands and mnemonics. When selected, only mnemonics and their operands are listed. When deselected, hexadecimal values are listed too.

TIP: To make it easier to use a disassembly listing for assembly language source code, select this item. With no hexadecimal values listed, disassembly data is easier to edit for use in assembly language source code.

Show Data Modules

Specifies whether or not to list data sections.

Disassemble Exception Tables

Specifies whether or not to list exception table data.

Show SYM Info

Specifies whether or not to list SYM (debugger) information in a disassembly listing.

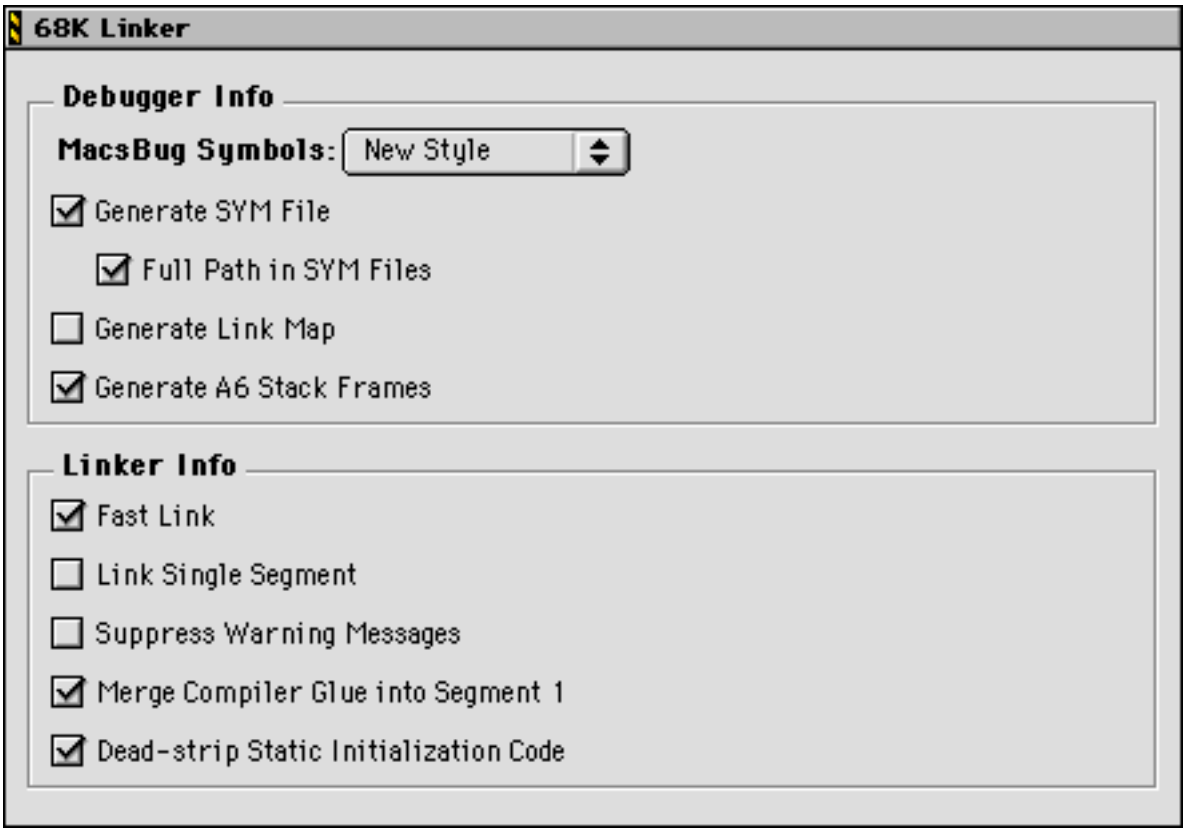
Show Name Table

Specifies whether or not to begin a disassembly with a list of all symbols in the object.

68K Linker

The 68K Linker panel, shown in Figure 14.15, is where you control settings related to linking your object code into final form, be it executable, library, or other type of code.

Figure 14.15 The 68K Linker panel.



These items in this panel are:

MacsBug Symbols
Generate SYM File
Full Path in SYM Files
Generate Link Map
Generate A6 Stack Frames
Fast Link

Link Single Segment	Suppress Warning Messages
Merge Compiler Glue into Segment 1	Dead-strip Static Initialization Code

MacsBug Symbols

Specifies what kind of debugger information to generate for MacsBug or Jasik's The Debugger.

Generate SYM File

Creates a SYM file for your project the next time it makes your project. To determine which source files CodeWarrior includes in that SYM file, use the Debug Info markers in the Project window. To learn how to set these markers, refer to the *IDE User Guide*.

Use `#pragma sym on` to generate SYM information for subsequent functions. Use `#pragma sym off` to stop generating SYM information.

Full Path in SYM Files

Puts the full pathname of the source file in the SYM file. Turning on this option is recommended. However, it slightly increases the size of your SYM file. If this option is off, the debugger might prompt you to locate a source file if it can't find it in the same folder as the SYM file that refers to it.

Generate Link Map

Create a text file that contains routine and class section information on the generated object code

Generate A6 Stack Frames

Specifies that each stack frame should set up and restore the A6 register. Many debuggers, including the IDE debugger, Jasik's The Debugger, and SourceBug™, require A6 stack frames, which let debuggers trace through the call stack and find each routine. If you are not planning to use a debugger, turn this option off to generate faster code.

These are the instructions that the compiler generates for each function, if this option is on:

```
LINK #nn,A6
. . . .
UNLK A6
```

This option corresponds to `#pragma a6frames`.

Fast Link

To speed up linking your project, use the Fast Link option. The trade-off for fast linking is that you use more memory, so the provides several speed levels to choose from. Turning on this option typically only has a noticeable improvement for large projects.

WARNING! If you are using function-level segmentation or MacApp in a 68K project, you *must* select **Fast Link**.

Link Single Segment

When selected, ignores all `#pragma segment` directives and the groups in the project window's **Segments** view, and puts all your code into one segment at link time when this option is checked. When deselected, generates a multi-segment code resource.

See 68K Target, Code Resource for more information on multi-segment code resources.

Suppress Warning Messages

Prevents the linker from reporting non-fatal warning messages. This option does not affect the warning messages displayed by other parts of the IDE, including compilers.

Merge Compiler Glue into Segment 1

Places compiler initialization instructions and inline C/C++ functions that can't be called inline into the first code segment.

Dead-strip Static Initialization Code

Removes unreferenced runtime initialization and object constructor instructions.

If the Dead-strip Static Initialization Code option is on, the linker removes runtime initializations and constructor calls for all the static and global objects in a file, if your program does not refer to any variables or functions in that file.

For example, suppose that a project contains the files A.cp, B.cp, and C.cp. The `main()` function in A.cp calls functions in B.cp but not in C.cp, and C.cp has some global variables of types which have constructors. The compiler generates code called a static initializer which calls the constructors for the variables in C.cp.

If this option is on, the linker recognizes that no references are made to any of C.cp's variables, so the linker does not link the variables or the static initializers for them into the program.

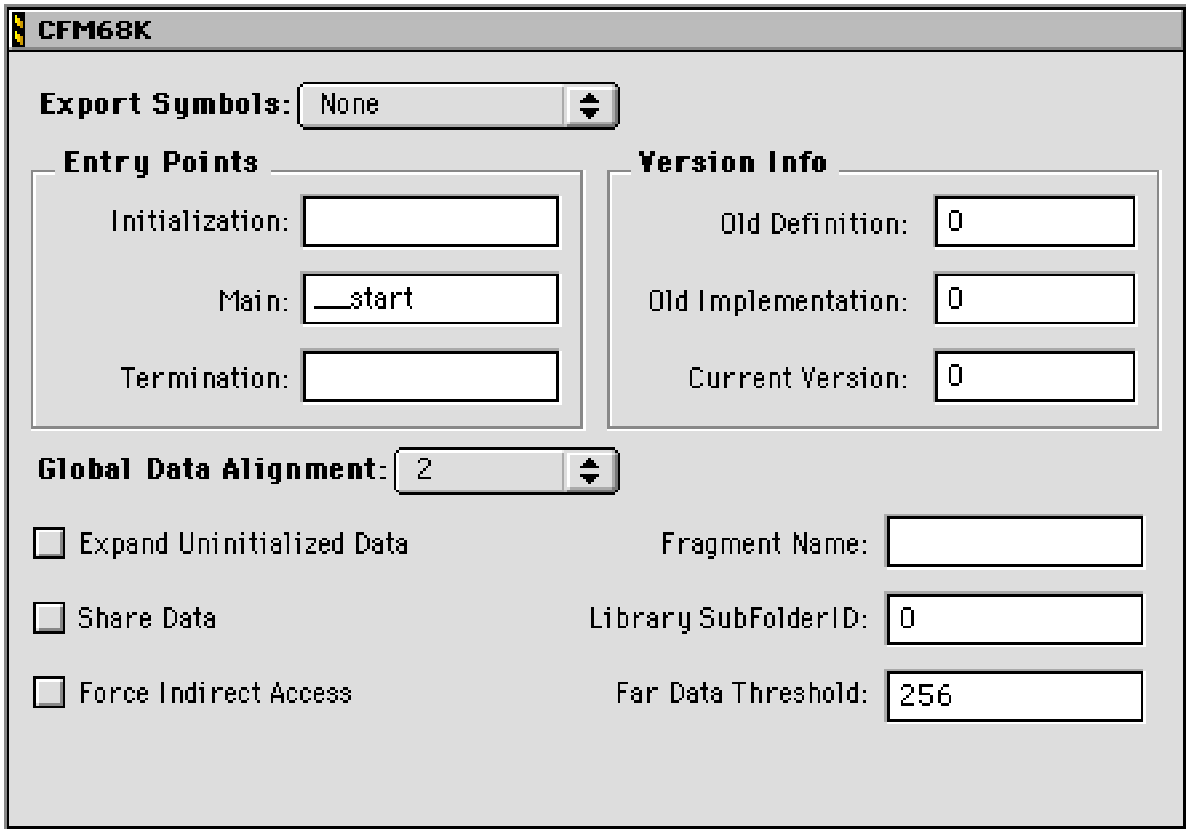
If this option is off, the linker leaves the variables and the static initializer in, and it may end up linking in variables and functions that the program never uses.

In general, leave this option on to produce smaller and faster code. However, if your program expects the static initializers to be executed for their side-effects, turn this option off.

CFM68K

The CFM68K panel, shown in , presents options for control the generation of a CFM68K fragment. For complete details about configuring and developing a CFM68K fragment, see *Inside Macintosh: Mac OS Runtime Architectures*.

Figure 14.16 The CFM68K panel



The items in this panel are:

- Export Symbols
- Initialization
- Main
- Termination
- Old Definition
- Old Implementation
- Current Version
- Global Data Alignment
- Expand Uninitialized Data
- Share Data
- Force Indirect Access
- Fragment Name
- Library SubFolderID
- Far Data Threshold

Export Symbols

Specifies where to look for a list of symbols to export from the CFM68K code fragment. Table 14.11 lists the options in this pop-up menu.

Table 14.11 Symbol export options

Choosing this option…	Tells the linker to
None	export no symbols
Use ".exp" file	export the symbols listed in a file ending with ".exp"
All globals	export all global symbols
Use #pragma	export all symbols enclosed between #pragma export on and #pragma export off directives

Initialization

Specifies the symbol name of the routine to call when the CFM68K fragment is loaded into memory. 68K calls this routine before referring to any other items in the container. Typically, the initialization routine prepares and sets up objects and services that the fragment will use when its main routine is called.

Main

Specifies the symbol name of the routine that is the fragment's main entry point.

Termination

Specifies the symbol name of the routine that 68K will call when the fragment's data instantiation is released. Typically, the termination routine deallocates and releases the objects and services that were prepared by the initialization routine.

Old Definition

Specifies the oldest acceptable version for a fragment which is called by this fragment.

Old Implementation

Specifies the old acceptable version number for a fragment which refers to an item in this fragment.

Current Version

Specifies the version of this fragment.

Global Data Alignment

Specifies the byte boundary that data structures are aligned to.

Expand Uninitialized Data

Specifies whether or not to expand unititialized data structures to their full size in the PEF container. If this option is selected, uninitialized objects are stored in uncompressed format in the PEF container. If this option is deselected, uninitialized objects are expanded to their full size at runtime only.

Share Data

Specifies whether or not the fragment's data is instantiated once. If selected, all processes that refer to the fragment use a single instantiation of the fragment's data. If deselected, a new instantiation of the fragment's data is created for each process that refers to the fragment.

Force Indirect Access

Specifies how the fragment accesses non-static objects. When selected, the fragment's non-static objects are treated as if they were from another fragment, making it possible to refer to objects without having them present at compile time.

Fragment Name

Specifies the name of the CFM68K code fragment. Leave this field blank if you want the code fragment name to be the same as the current target's output name. See "68K Target, CFM68K Application" on page 261 and "68K Target, CFM68K Shared Library" on page 270 for information on specifying the output name.

Library SubFolderID

Specifies the resource ID of the alias resource that specifies where to search for libraries referred to by this code fragment.

Far Data Threshold

Specifies the size, in bytes, of data items that will be stored in far data space. Items that are this size or greater will be stored in far data space. Otherwise, items are stored in near data space.

Debugger Settings

The Debugger Target Settings panel presents options for selecting the tools to use for debugging your project.

The options in this panel are:

Log System Messages

Program to Launch for Debugging

Log System Messages

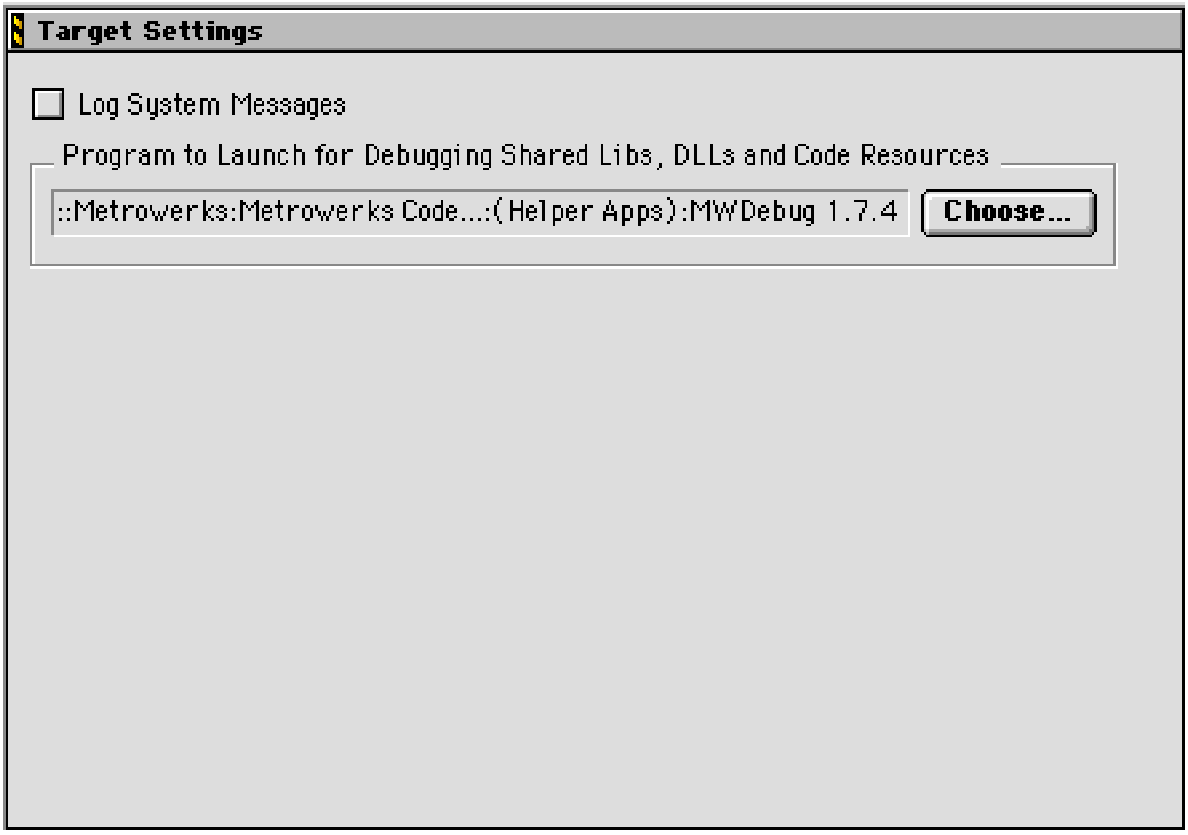
Controls the display of messages in the debugger's log window. If selected, messages about the actions the debugger takes while controlling your application appear in the log window.

NOTE: This feature is only available when debugging on Windows.

Program to Launch for Debugging

Launches a program before running the project under the debugger's control. Click the Choose button to select a program. This option is most useful when debugging a piece of software that acts as a plugin or extension to another program.

Figure 14.17 Debugger Target Settings panel


Target Settings
Log System Messages
Program to Launch for Debugging Shared Libs, DLLs and Code Resources
::Metrowerks:Metrowerks Code...:(Helper Apps):MWDebug 1.7.4
Choose...

15

C and C++ for 68K

This chapter describes both the CodeWarrior back-end compiler and linker for 68K, including support for inline assembly in the compiler.

The sections in this chapter are:

- Language Extensions for 68K
- Number Formats for 68K
- Calling Conventions for 68K
- Variable Allocation for 68K
- Register Variables for 68K
- Optimizing Code for 68K
- Generating Code for Specific 68K Processors
- Pragmas for 68K
- Linker Issues for 68K

However, this chapter does *not* discuss several important topics with respect to code generation, including front-end compiler issues, compiler and linker errors, controlling the size of C++ code, and so forth. These topics are covered in other CodeWarrior documentation as outlined in Table 15.1.

Table 15.1 Other compiler/linker documentation

For this topic...	See...
how CodeWarrior implements the C/C++ language	*C Compilers Reference* generally
using C/C++ Language and C/C++ Warnings settings panels	*C Compilers Reference*, "Setting C/C++ Compiler Options" chapter

For this topic...	See...
controlling the size of C++ code	*C Compilers Reference*, "C++ and Embedded Systems" chapter
using compiler pragmas	*C Compilers Reference*, "Pragmas and Symbols" chapter
initiating a build, controlling which files are compiled, handling error reports	*IDE User Guide*, "Compiling and Linking" chapter
information about a particular error	*Error Reference*
68K assembler	*Assembler Guide*

NOTE: Some of the items discussed in this chapter may actually be implemented in the front-end compiler. However, it really doesn't matter whether the actual implementation of a feature occurs in the front-end or back-end compiler. From the programmer's point of view, it is all one compiler.

Language Extensions for 68K

This section describes the 68K-specific extensions to the C and C++ standards found in the CodeWarrior C/C++ compiler.

You can disable some of these extensions with options in the C/C++ Language panel. This panel is described fully in the *C Compilers Reference* manual.

The topics in this section are:

- Inline Data—how to include simple inline data with the `asm` declaration.
- Specifying the Registers for Arguments—how to specify which registers a function uses for its parameters and the return value.

- PC-Relative Strings—how to control where the compiler stores strings.

Inline Data

When targeting 68K, the C/C++ compiler let you include simple inline data with the asm declaration. Use this syntax:

```
asm { constant, constant, . . . }
asm ( constant, constant, . . . )
```

A *constant* can be a numeric constant or a string literal.

For example, this function:

Listing 15.1 Inline data example

```
void foo()
{
  asm ( (short)0x4e71,(short)0x4e71 );
    // two 68K NOP instructions

  asm { 0x4e714e71,0x4e714e71 };
    // four 68K NOP instructions

  asm ((char)'C',(char)'o',(short)'de',"Warrior");
}
```

Produces assembly code that looks like this:

Listing 15.2 Assembly code from inline data

```
  LINK   A6, #$0000
  NOP
  NOP                       ; First two NOPs
  NOP
  NOP
  NOP
  NOP                       ; Next four NOPs
```

```
  DC.B    "CodeWarrior\0"
  UNLK    A6
  RTS
```

Specifying the Registers for Arguments

(K&R, §A8.6.3, §A10.1) When targeting 68K, the C/C++ compiler lets you specify which registers a function uses for its parameters and return value. Registers `D0-D2`, `A0-A1`, and `FP0-FP3` are available.

When you declare the function, specify the registers by using the `#pragma parameter` statement before the declaration. When you define the function, specify the registers right in the argument list.

This is the syntax for the `#pragma parameter`:

```
#pragma parameter return-reg func-name(param-regs)
```

The compiler passes the parameters for the function *func-name* in the registers specified in *param-regs* instead of the stack, and returns any return value in the register *return-reg*. Both *return-reg* and *param-regs* are optional.

For example, Listing 15.3 shows the declaration and definition of a function, in which `a` is passed in `D0`, `p` is passed in `A1`, `x` is passed in `FP0`, `f` is passed on the stack, and the return value is in D2.

Listing 15.3 Using registers with functions

```
// prototype
#pragma parameter __D2 function(__D0,__A1,__FP0)
short function(long a, Ptr p, long double x, short f);

// function definition
short function(long a:__D0, Ptr p:__A1,
               long double x:__FP0, short f) :__D2
{
```

```
  // ...
}
```

PC-Relative Strings

If the **PC-Relative Strings** option in the 68K Processor settings panel is on, the compiler stores the string constants used in a local scope in the code segment and addresses these strings with PC-relative instructions. If this option is off, the compiler stores all string constants in the global data segment. This option helps keep your global data segment smaller.

Regardless of how this option is set, the compiler stores string constants that have global scope in the global data segment. Listing 15.4 shows an example.

Listing 15.4 Using PC-relative strings

```
#pragma pcrelstrings on

int f(char *);

int x = f("Hello"); // "Hello" allocated in global data segment

int bar()
{
  return f("World"); // "World" allocated in code segment
                     // (pc-relative)
}
#pragma pcrelstrings reset
```

NOTE: If you turn the **Pool Strings** option on, the compiler ignores the setting of the **PC-Relative** Strings option.

The **PC-Relative Strings** option corresponds to the pragma `pcrelstrings`. To check whether this option is on, use `__option(pcrelstrings)`. By default, this option is off.

Number Formats for 68K

This section describes how the CodeWarrior C/C++ compilers implement integer and floating-point types for 68K processors. You can also read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types.

The topics in this section are:

- 68K Integer Formats
- 68K Floating Point Formats

68K Integer Formats

The 68K back-end compiler lets you choose the number of bytes allocated for an `int`. Use the **4-Byte Ints** option in the 68K Processor settings panel.

In general, you'll turn this option on for Mac OS programming because it's easier to port your code to the PowerPC compiler (which always uses 4-byte ints). However, 2-byte ints are slightly more efficient on the 68K, so you may want to turn this option off when efficiency is more important.

Table 15.2 shows the size and range of the integer types available when targeting 68K.

Table 15.2 68K integer types

For this type	Option setting	size is	and its range is
bool	n/a	8 bits	`true` or `false`
`char`	**Use Unsigned Chars** is off in the C/C++ Language panel	8 bits	`-128` to `127`
`char`	**Use Unsigned Chars** is on in the C/C++ Language panel	8 bits	`0` to `255`

For this type	Option setting	size is	and its range is
signed char	n/a	8 bits	`-128` to `127`
`unsigned char`	n/a	8 bits	`0` to `255`
short	n/a	16 bits	`-32,768` to `32,767`
`unsigned short`	n/a	16 bits	`0` to `65,535`
`int`	**4-Byte Ints** is off in the 68K Processor panel	16 bits	`-32,768` to `32,767`
`int`	**4-Byte Ints** is on in the 68K Processor panel	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned int`	**4-Byte Ints** is off in the 68K Processor panel	16 bits	`0` to `65,535`
`unsigned int`	**4-Byte Ints** is on in the 68K Processor panel	32 bits	`0` to `4,294,967,295`
`long`	n/a	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned long`	n/a	32 bits	`0` to `4,294,967,295`
long long	n/a	64 bits	`-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`
unsigned long long	n/a	64 bits	`0` to `18,446,744,073,709,551,615`

68K Floating Point Formats

The 68K back-end compiler lets you choose the number of bytes allocated for a `double`. Use the **8-Byte Doubles** option in the 68K Processor settings panel.

In general, turn this option off because 8-byte (64-bit) doubles are less efficient. However, if you are porting code that relies on 8-byte doubles, turn this option on.

You can also choose to create code that is optimized for machines with a 68040 processor or a 68881 floating point unit. If you turn on the **68881 Codegen** option in the 68K Processor settings panel, the compiler uses floating-point operations and types that are designed specifically for those chips.

Turn on the **68881 Codegen** option only if the code contains lots of floating-point operations, must be as fast as possible, and you're sure the code will be used only on machines that contain a 68040 or 68881. Table 15.3 shows the size and range of the floating-point types available when targeting a 68K processor.

WARNING! If you create program code with the **68881 Codegen** option on and try to run it on a machine that does not have a 68040 or 68881, the code will crash. Use the Mac OS Gestalt Manager for information on detecting the presence of a 68040 or 68881 processor at run time.

Table 15.3 68K floating point types

For this type	Option setting	Size is	and its range is
`float`	n/a	32 bits	`1.17549e-38` to `3.40282e+38`
short double	n/a	64 bits	`2.22507e-308` to `1.79769e+308`
double	8-Byte Doubles is on	64 bits	`2.22507e-308` to `1.79769e+308`
	8-Byte Doubles is off and 68881 Codegen is off	80 bits	`3.362103e-4932` to `1.18973e+4932`
	8-Byte Doubles is off and 68881 Codegen is on	96 bits	`1.68105e-4932` to `1.18973e+4932`

For this type	Option setting	Size is	and its range is
long double	68881 Codegen is off	80 bits	`3.362103e-4932` to `1.18973e+4932`
	68881 Codegen is on	96 bits	`1.68105e-4932` to `1.18973e+4932`

Calling Conventions for 68K

This section describes the C/C++ calling conventions for 68K development.

The compiler passes all parameters on the stack in reverse order.

The compiler passes the return value in different locations, depending on the nature of the value and compiler settings.

- It returns an integer value in register `D0`.
- It returns a pointer value in register `A0`.
- If it returns a value of any other type, the caller reserves temporary storage area for that type in the caller's stack and passes a pointer to that area as the last argument. The callee returns its value in the temporary storage area.

There are two options that change how the compiler returns a value.

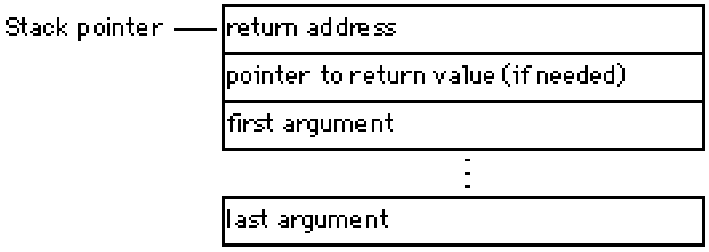
- If you turn on either the pragma `pointers_in_D0` or pragma `mpwc`, the compiler returns pointer values in register `D0`. Use one of these pragmas if you're calling a function declared in an MPW library.
- If the **68881 Codegen** option is on, the compiler returns 96-bit floating-point values in register FP0.

WARNING! If you create program code with the **68881 Codegen** option on and try to run it on a machine that does not have a 68040 or 68881, the code will crash. Use the Mac OS Gestalt

Manager for information on detecting the presence of a 68040 or 68881 processor at run time.

Figure 15.1 shows what the stack looks like when you call a C function with the 68K compiler.

Figure 15.1 Calling a C function



Variable Allocation for 68K

(K&R, §A4.3, §A8.3, §A8.6.2) This section describes how the C/C++ compiler allocates space for variables.When targeting 68K, the C/C++ compiler lets you declare structs and arrays to be any size. However, it does place some limits on how you allocate space for them. Also, the operating system your software will run on may also limit the size of variable allocations.

Bitfields can be only 32 bits or less.

A function cannot contain more than 32K of local variables. To avoid this problem, do one of the following:

- Dynamically allocate large variables.
- Declare large variables to be `static`. Note that you may run into the 32K limit on global variables, described below.

You cannot declare a global variable that is larger than 32K unless you use `far` data. You must do one of the following:

- Dynamically allocate the variable.
- Use the `far` qualifier when declaring the variable.
- Turn on the **Far Data** option in the 68K Processor settings panel or use the pragma `far_data`.

  Listing 15.5 below shows an example of how to declare a large struct or array.

Listing 15.5 Declaring a large structure

```
int i[50000]; // Wrong with 68K compiler and the Far Data
              // option in the Processor settings panel is off

far int j[50000]; // ALWAYS OK.

int *k;
&k = malloc(50000 * sizeof(int));  // ALWAYS OK.
```

Register Variables for 68K

(K&R, §A4.1, §A8.1) This section describes how the C/C++ compilers allocate variables to registers. The 68K back-end compiler automatically allocates local variables and parameters to registers according to how frequently they're used and how many registers are available. If you're optimizing for speed, the compiler give preference to variables used in loops.

The 68K back-end compiler gives preference to variables declared to be `register`, but does not automatically assign them to registers. For example, if the compiler has to choose between a variable from an inner loop and a variable declared `register`, the variable from the inner loop will be placed in the register.

The 68K compiler can use these registers for local variables:

- `A2` through `A4` for pointers
- `D3` through `D7` for integers and pointers.

If you turn on the **68881 Codegen** option, the 68K compilers also use these registers:

- FP4 through FP7 for 96-bit floating-point numbers

WARNING! If you create program code with the **68881 Codegen** option on and try to run it on a machine that does not have a 68040 or 68881, the code will crash. Use the Mac OS Gestalt Manager for information on detecting the presence of a 68040 or 68881 processor at run time.

Optimizing Code for 68K

This section discusses optimizations that are specific to 68K development with CodeWarrior. This topic is:

- Register Coloring

More optimizations are available through the **Global Optimizations** panel.

Register Coloring

The C/C++ compiler can perform an optimization called *register coloring*. In this optimization, the compiler assigns two or more variables to the same register. It does this if the source code does not use the variables at the same time. In this example, the compiler could place `i` and `j` in the same register:

```
short i;
int j;

for (i=0; i<100; i++) { MyFunc(i); }
for (j=0; j<100; j++) { MyFunc(j); }
```

However, if a line of code like the one below appears anywhere in the function, the compiler would realize that you are using `i` and `j` at the same time, and place them in different registers.

```
MyFunc (i + j);
```

Register coloring reduces code size and has no effect on execution time.

If register coloring is on while you debug your code, it may appear as though there's something wrong with the variables that share a single register. In the example above, `i` and `j` would always have the same value. When `i` changes, `j` changes in the same way, and vice versa.

You can prevent this by turning off register coloring, or by declaring the variables as `volatile`.

You control whether the 68K back-end compiler performs the register coloring optimization from the **Global Optimizations** settings panel. Refer to the *IDE User Guide* for information on this panel.

The **Global Register Allocation** option corresponds to the pragma `no_register_coloring`. To check whether this option is on, use `__option (no_register_coloring)`. By default, this option is off.s

Generating Code for Specific 68K Processors

This section describes how to use the CodeWarrior compiler to generate code for a specific processor. You can specify a processor in the **68K Processor** panel. For a full discussion of this panel and its options, see "68K Processor" on page 278.

When targeting 68K processors, the compiler can generate code for specific 68K processors: the MC68020 processor, and the MC68881 floating-point unit.

You control whether the 68K back-end compiler generates code for specific 68K processors with options in the **68K Processor** settings panel. The options are the **68020 Codegen** checkbox and the **68881 Codegen** option in the **Floating Point** pop-up menu.

WARNING! If you create program code with the **68020 Codegen** or **68881 Codegen** options on and try to run it on a machine that does not have the corresponding processor, the code will crash. Use the Mac OS Gestalt Manager for information on detecting the presence of a proper processor at run time.

You should use these options only if your application will run solely on machines that have that specific processor and your application needs the extra features that the processor provides. Code compiled for a specific 68K processor may not work on another 68K processor.

This sections contains the following topics:

- Generating Code for the MC68000
- Generating Code for the MC68020
- Generating Code for the MC68881
- Hazards of 68020 and 68881 Codegen

Generating Code for the MC68000

The C/C++ compiler generates object code for the Motorola MC68000 processor if the **68020 Codegen** checkbox is disabled and **6881 Codegen** is not selected from the **Floating Point** pop-up menu in the **68K Processor** settings panel.

When generating 68000 object code, keep in mind that functions are limited to 32K in size. CodeWarrior C/C++ uses relative branching, and the MC68000's relative branching instructions are limited to 16 bits. Only MC68020 and higher processors have 32-bit relative branching instructions, which the C/C++ compiler will generate if **68020 Codegen** is on.

Generating Code for the MC68020

The C/C++ compiler lets you take full advantage of the MC68020 processor. This feature is controlled by the **68020 Codegen** checkbox in the **68K Processor** settings panel. For information on this panel, see "68K Processor" on page 278.

To generate code optimized for the 68020 processor, select the **68020 Codegen** checkbox on the **68k Processor** settings panel. When you enable this option, the C/C++ compiler use the extensions available in the MC68020 instruction set, including integer multiplication, integer division, and bit-field operations.

Generating code for 68020 and higher processors makes it possible to have functions that are greater than 32K, thanks to the 32-bit relative addressing instructions of these processors.

However, think carefully before you use the **68020 Codegen** option. Your code will not run on any machine that does not have an MC68020 or equivalent, including a Mac OS computer using a PowerPC chip. See "Hazards of 68020 and 68881 Codegen" on page 317.

The **68020 Codegen** option corresponds to the pragma `code68020`. To check whether this option is on, use `__option (code68020)`. By default, this option is off.

Generating Code for the MC68881

The C/C++ compiler lets you take full advantage of the MC68881 floating-point unit (FPU). This feature is controlled by the **68881 Codegen** option on the **Floating Point** pop-menu on the **68K Processor** settings panel.

The MC68881 is built into most versions of the MC68040 processor, and it is included separately in many Macintosh computers that contain the MC68030 or MC68020 processor. Think carefully before you choose the **68881 Codegen** option. Your code will not run on any machine that does not have an MC68881, including a Mac OS computer using a PowerPC chip or a 68K chip that does not include the 68881 FPU. See "Hazards of 68020 and 68881 Codegen" on page 317.

To generate code optimized for the 68881 FPU, choose the **68881 Codegen** option. When you choose it, the C/C++ compiler generates code optimized for the MC68881.

It stores variables declared `long double` or `extended` in 96 bits. It uses MC68881 instructions for basic arithmetic operations, such as

addition, subtraction, multiplication, division, and comparisons. The header files `fp.h` and `math.h` use MC68881 instructions for many transcendental and floating-point conversions. The compiled code is faster and computes the same results as code compiled with the option off.

NOTE: If you are programming for Mac OS, you should be aware that the Mac OS Toolbox uses the MC68881 for many floating-point functions. You do not need to choose **68881 Codegen** from the **Floating Point** menu on the **68K Processor** panel to take advantage of the FPU indirectly through the Toolbox.

The **68881 Codegen** option corresponds to the pragma `code68881`. To check whether this option is on, use `__option (code68881)`. By default, this option is off.

The rest of this section describes what happens when you choose **MC68881 Codegen**.

Using the Extended data type

If you choose the **68881 Codegen** option, the compiler stores any variable declared `extended` or `long double` in the Motorola 96-bit format, instead of the SANE 80-bit format. Both formats meet the IEEE standards for accuracy. The main difference between them is that the 96-bit format contains 16 bits of padding so that an extended number fits evenly into three 32-bit memory accesses.

`MacTypes.h` defines the `extended` type. `SANE.h` contains two other type definitions: `extended80` and `extended96`. It also contains functions that convert between 80-bit and 96-bit formats: `x96tox80()` and `x80tox96()`.

Using floating-point registers

The MC68881 has eight registers, FP0 through FP7, that store 96-bit floating-point values (that is, `extended` or `long double`). If you choose the **68881 Codegen** option, your assembly language routines can use registers FP0 through FP3 for temporary storage without re-

storing their values. If you use registers FP4 through FP7, you must preserve their contents.

The compiler allocates variables of type `long double` or `extended` to registers to optimize performance.

Hazards of 68020 and 68881 Codegen

You can compile code specifically aimed at either the MC68020 or the MC68881 FPU. For details, see "Generating Code for the MC68020" on page 314 and "Generating Code for the MC68881" on page 315.

However, you should use these options only if your application will run solely on machines that have that specific processor, and your application needs the extra features that the processor provides. If you know your code will only be run on a particular machine with the required chip, there is no problem.

For general Mac OS programming, you probably can't be sure. Code compiled for a specific 68K processor may not work on another 68K processor, and will not work on a PowerPC chip. If your code *requires* chip-specific features, you should test for the presence of the chip and alert the user to its absence. Use the Mac OS Toolbox `Gestalt()` function to make sure the chip is available. For more information on `Gestalt()`, see the "Gestalt Manager" chapter in *Inside Macintosh: Operating System Utilities*.

Listing 15.6 shows source code that displays an alert if the application is for an MC68881 and the machine does not have an MC68881.

Listing 15.6 Checking for the MC68881 at runtime

```
void main(void)
{
if (!HasFPU())    // Calls gestalt() to check
  {                 //  if the computer has FPU
    DisplayNoFPU(); // Displays an alert
    return;         //  saying there is no FPU
  }
```

```
// . . .
}
```

Note that `HasFPU()` and `DisplayNoFPU()` are not Toolbox functions. If you use this code, you must define these functions so that they accomplish the required task—use `Gestalt()` to determine the presence or absence of the chip, and display an appropriate alert.

If you would like to take advantage of chip-specific features, but your code will work without them, then you should create alternative code for the presence or absence of the chip. You would use `Gestalt()` in the same way to test for the chip, and then use its presence or absence to determine which way your code works.

You can create alternative builds within your code to compile for different targets. Use the `__option()` pre-processor function. Use `__option(code68881)` to check whether the **68881 Codegen** option is on. Use `__option(code68020)` to check whether the **68020 Codegen** option is on.

Listing 15.7 shows an example that uses different code depending on whether you are generating code for the MC68881.

Listing 15.7 Checking for the MC68881 at compile time

```
int calc(double i)
{
#if __option (code68881)
  // generate code optimized for the floating point unit
#else
  // generate code for any 68K processor
#endif
}
```

Pragmas for 68K

Table 15.4 lists the pragmas supported for 68K development.

Table 15.4 Pragmas for 68K development

a6frames	exceptions	mpwc_relax	side_effects
align	extended_error check	once	SOMCallStyle
align_array_ members	far_data	only_std_ keywords	SOMCheck-Environment
ANSI_strict	far_vtables	optimize_for_ size	SOMMetaClass
ARM_conform	force_active	parameter	SOMReleaseOrder
auto_inline	IEEEdoubles	pcrelstrings	static_inlines
bool	ignore_oldstyle	pointers_ in_A0, pointers_in_D0	toc_data
code68020	inline_depth	pool_strings	unsigned_char
cplusplus	lib_export	precompile_ target	unused
cpp_extensions	longlong	profile	warn_emptydecl
direct_ destruction	macsbug, oldstyle_ symbols	require_ prototypes	warn_ hidevirtual
dont_reuse_ strings	mark	RTTI	warn_illpragma
enumsalwaysints	mpwc_newline	segment	warn_unusedarg

These pragmas, pragma syntax, and how to determine and modify the state of the compiler using pragmas are all detailed in the *C Compilers Reference*.

Linker Issues for 68K

This section discusses the background information on the 68K linker and how it works. The topics in this section are:

- Deadstripping Unused Code and Data
- Link Order

Deadstripping Unused Code and Data

The 68K linker deadstrips unused code and data only from files compiled by the CodeWarrior C/C++ compiler. Assembler relocatable files and C/C++ object files built by other compilers are never deadstripped. Deadstripping is particularly useful for C++ programs. Libraries (archives) built with the CodeWarrior C/C++ compiler only contribute the used objects to the linked program. If a library has assembly or other C/C++ compiler built files, only those files that have at least one referenced object contribute to the linked program. Completely unreferenced object files are always ignored.

Link Order

Link order is generally specified in the Segments view of the Project window. For general information on setting link order, see the *IDE User Guide*.

Regardless of the link order specified in the Segments view of the Project window, the 68K linker always processes C/C++ or assembler source files before it processes relocatable files (.o) or archive files (.a), which are treated as libraries. Therefore if a symbol is defined in a source file, the linker will use that definition in preference to a definition in a library.

There is one exception. If the source file definition is a weak symbol, then the linker will use a global symbol in a library. You can create a weak symbol with `#pragma overload`.

The 68K linker ignores executable files that are in the project. You may find it convenient to keep the executable there so that you can disassemble it. If a build is successful, the file will show up in the

project as out of date (there will be a check mark in the touch column on the left side of the project window) because it is a new file. If a build is unsuccessful, the IDE won' t be able to find the executable file and will stop the build with an appropriate message.

16

Pascal for 68K

This chapter covers topics for using the CodeWarrior Pascal compiler to develop software running on Motorola 680x0 processors.

- Language Extensions for 68K
- Data Type Formats for 68K
- Calling Conventions for 68K

For other topics on code generation issues see these topics:

- "Optimizing Code for 68K" on page 312
- "Generating Code for Specific 68K Processors" on page 313
- "Linker Issues for 68K" on page 320

This chapter does *not* discuss several important topics with respect to code generation, including front-end compiler issues, compiler and linker errors, and so forth. These topics are covered in other CodeWarrior documentation, shown in Table 16.1.

Table 16.1 Other compiler/linker documentation

For this topic...	See...
how CodeWarrior implements the Pascal language	*Pascal Compilers Reference* and *Pascal Language Reference*
using the **Pascal Language** and **Pascal Warnings** settings panels	*Pascal Compilers Reference*, "Setting Pascal Compiler Options" chapter
using compiler pragmas	*Pascal Compilers Reference*, "Pragmas and Symbols" chapter

For this topic...	See...
initiating a build, controlling which files are compiled, handling error reports	*IDE User Guide*, "Compiling and Linking" chapter
information about a particular error	*Error Reference*
68K assembler	*Assembler Guide*

NOTE: Some of the items discussed in this chapter may actually be implemented in the front-end compiler. However, it really doesn't matter whether the actual implementation of a feature occurs in the front-end or back-end compiler. From the programmer's point of view, it is all one compiler.

Language Extensions for 68K

This section describes elements of the CodeWarrior Pascal compiler implemented specifically for the 68K platform. At this time there is one 68K-specific extension, the `INLINE` directive.

The INLINE Directive

The `INLINE` directive allows you to write explicit hexadecimal 68K machine instructions in place of the block. The code is expressed in constants or constant expressions.

When a normal procedure or function is called, the compiler generates code that pushes the procedure's arguments on the stack (along with two or four bytes for a return value, if this is a function) and then generates an assembly-language `JSR` (Jump to SubRoutine) to call the procedure. When you use a procedure declared `INLINE`, CodeWarrior Pascal generates code (in place of the `JSR`) from the constants following the word `INLINE`.

Each constant (or constant expression) represents exactly one machine-instruction word in the code generated by the compiler. The code is generated in the order of the constants. Take care that you observe the proper rules for adjusting the stack, saving registers, and so on. These are documented in *Inside Macintosh* and in the "Pascal and C" chapter of the *Pascal Language Reference.*

This facility is intended for writing small routines and calls to OS routines ("glue code"). If you want to use large amounts of assembly code, create an external procedure instead.

NOTE: No block is ever defined in an `INLINE` directive. `INLINE` can also be used in the interface part of a unit. In that case, there is still no block for the procedure in the corresponding implementation part.

Here is an example of a procedure declared `INLINE`:

```
PROCEDURE trap (Tos: LONGINT); INLINE $A9ED;
```

WARNING! The @ operator cannot be used to generate a pointer to an `INLINE` routine.

Data Type Formats for 68K

This section describes how the CodeWarrior Pascal compiler implements classes of standard Pascal data types for 68K. These classes are:

- Integer Types for 68K
- Floating Point Types for 68K
- Array Types and PACKED Arrays for 68K
- Character String Types for 68K
- Sets for 68K
- Enumerations for 68K

- Records for 68K

Integer Types for 68K

This section describes how `INTEGER`, `LONGINT` and a few other atomic data types are implemented for 68K in CodeWarrior Pascal.

`INTEGER` for 68K

The `INTEGER` data type on the 68K platform is a 16-bit, 2's-complement integer. The range of the type `INTEGER` is the set of values

```
-(MaxINT+1), -MaxINT,...,
-1, 0, 1, ... MaxINT-1, MaxINT
```

that is, –32768 to +32767.

`LONGINT` for 68K

The `LONGINT` data type on the 68K platform is a 32-bit 2's-complement integer. The range of the type `LONGINT` is the set of values

```
-(MaxLONGINT+1), -MaxLONGINT, ...-1, 0, 1,
... MaxLONGINT-1, MaxLONGINT
```

that is, –2147483648 to +2147483647.

Arithmetic on `INTEGER` and `LONGINT` operands is done in both 16-bit and 32-bit precision, as follows:

- All integer constants in the range of type `INTEGER` are considered to be of type `INTEGER`. All integer constants in the range of type `LONGINT`, but not in the range of type `INTEGER`, are considered to be of type `LONGINT`.
- When both operands of an operator (or the single operand of a unary operator) are of type `INTEGER`, 16-bit operations are always performed and the result is of type `INTEGER` (truncated to 16 bits if necessary). Similarly, if both operands are of type `LONGINT`, 32-bit operations are always performed and the result is of type `LONGINT`.
- When one operand is of type `LONGINT` and the other is of type `INTEGER`, the `INTEGER` operand is converted to `LONG-`

`INT`, 32-bit operations are performed, and the result is of type `LONGINT`. However, if this value is assigned to a variable of type `INTEGER`, it is truncated (see next rule).

- The expression on the right of an assignment statement is evaluated independently of the size of the variable on the left. For example, if variable `longVar` is declared as type `LONGINT`, the statement `longVar := MaxINT+MaxINT` will still cause `INTEGER` overflow. If necessary, the result of the expression is truncated or extended to match the size of the variable on the left.

An important point to remember is that each operator is applied only to its two operands, so, at most, one of those operands is converted. If the expression contains other operands, those are not necessarily converted. For example, in the expression

```
oneInt+twoInt+threeInt+oneLongint
```

the value of `oneInt` is added to `twoInt` in a 16-bit operation, the result is added to `threeInt` in another 16-bit operation, and the result of that is converted to a `LONGINT` value and added to `oneLongint`. The result of the expression is a `LONGINT`.

The `Ord4` function described in the Pascal Library Reference can be used to convert an `INTEGER` value to a `LONGINT` value.

TIP: For the 68000-family processors, operations other than division and multiplication on `LONGINT` values take approximately one and a half times as long as corresponding operations on `INTEGER` values. Division and multiplication take more than twice as long.

`UNSIGNEDBYTE` for 68K

The CodeWarrior Pascal for 68K compiler implements this type as an 8-bit unsigned integer for the 68K platform. The range of the type `UNSIGNEDBYTE` is the set of values 0 to +255.

This type can be used in expressions with the `UNSIGNEDWORD`, `UNSIGNEDLONG`, `INTEGER` and `LONGINT` types, but no range checking is available.

UNSIGNEDWORD for 68K

The CodeWarrior Pascal compiler implements this type as a 16-bit unsigned integer for the 68K platform. The range of the type UNSIGNEDWORD is the set of values

```
0, 1, ..., 2 * MaxINT, (2 * MaxINT ) + 1
```

that is, 0 to +65535.

This type can be used in expressions with the UNSIGNEDLONG, INTEGER and LONGINT types, but no range checking is available.

UNSIGNEDLONG for 68K

The CodeWarrior Pascal compiler implements this type as a 32-bit unsigned integer for the 68K platform. The range of the type UNSIGNEDLONG is the set of values

```
0, 1, ..., 2 * MaxLONGINT, (2 * MaxLONGINT ) + 1
```

that is, 0 to +4294967295.

This type can be used in expressions with the UNSIGNEDWORD, INTEGER and LONGINT types, but no range checking is available.

Floating Point Types for 68K

There are four floating-point types in CodeWarrior Pascal, all predefined:

- REAL for 68K
- SINGLE for 68K
- DOUBLE for 68K
- LONGREAL for 68K

For 68K there is also the EXTENDED for 68K floating-point type.

The ANS REAL for 68K type is identical to the SINGLE type. CodeWarrior Pascal accepts both identifiers and treats them identically.

REAL for 68K

For 68K, this type takes up four bytes of storage (32 bits). The range of this type is approximately 1.401298464E–45 to 3.402823466E38 in scientific notation. REAL has 7 or 8 digits of precision.

SINGLE for 68K

This type is the same as REAL for 68K.

LONGREAL for 68K

For the 68K platform, this type takes 8, 10, or 12 bytes of storage depending on whether or not the IEEE doubles option in the 68K Processor preferences panel is active.

When selected, LONGREALs are stored as 8-byte doubles. When the IEEE doubles option is deselected, LONGREALs are stored using 12 bytes (if the **68881 Codegen** option in the 68K Processor preferences panel or the 68881 directive is turned on), or 10 bytes (when the 68881 Codegen option is deselected).

NOTE: The 8-byte doubles option takes precedence over the 68881 codegen option.

DOUBLE for 68K

This type is the same as LONGREAL for 68K.

EXTENDED for 68K

EXTENDED is similar to LONGREAL except that the minimum size of a variable declared EXTENDED is 10 bytes. Variables declared EXTENDED can also be 12 bytes long.

To be able to declare EXTENDED variables without the compiler issuing a warning, the IEEE doubles option in the **68K Processor** settings panel *must* be deselected or the equivalent pragma turned on.

What determines whether `EXTENDED` are 10 or 12 bytes long is whether the 68881 Codegen option in the same preferences panel, or the equivalent pragma, is

- Selected / turned on: `EXTENDED` are 12 bytes in size
- Deselected / turned off: `EXTENDED` are 10 bytes in size

Array Types and PACKED Arrays for 68K

This section documents anything distinctive about how array and PACKED array types are implemented for 68K in CodeWarrior Pascal.

Array Types for 68K

For the 68K platform, CodeWarrior Pascal implements arrays using even padded elements. That is, the compiler ensures that each array element begins at an address or offset that is divisible by 2.

PACKED Arrays for 68K

PACKED arrays on 68K are implemented like standard arrays, using even padded elements, but the addresses or offsets will not necessarily be divisible by 2 depending on what you have selected for the **Struct Alignment** option in your 68K project. The divisor will be:

- 2 if **Struct Alignment** is set to **68K**
- 4 if **Struct Alignment** is set to **68K 4-byte**
- 4 or 8 if **Struct Alignment** is set to **PowerPC**, depending on the size of the array elements. The compiler makes the determination on-the-fly.

Also, PACKED arrays of characters use 8 bits per character for storage, not the 68K-standard 16 bits.

Character String Types for 68K

This section documents anything distinctive about how character string types are implemented for 68K in CodeWarrior Pascal.

Pascal and STRING for 68K

For 68K, a Pascal string is stored as a one-byte-length field followed by the characters in the string. You can therefore change the length of the string by changing its zeroth character. For example,

```
myStr[0] := Chr(Ord(myStr[0])+n);
```

changes the length of `myStr` by the value of `n`.

C and PACKED ARRAY OF CHAR for 68K

In CodeWarrior Pascal, when you declare a string constant using double quotation marks (") rather than Pascal-standard single quotation marks ('), that string is stored as a C-style string. That means the first byte of the string does not contain the string's length, and it is terminated with an ASCII 0 (NUL) character.

Care is required when assigning C-style string constants to other variables and passing them in procedures and functions:

- If you want `destination` in

  ```
  destination := CStyleStringConstant
  ```

  to also be stored as a C-style string you must declare it as a PACKED array of char. This is also true when `destination` is a procedure or function parameter to which the value of a C-style string constant will be assigned.

  If `destination` is declared as an ordinary Pascal string, after assignment it will contain the standard prefixing length byte and the first 255 bytes of the C-style string, and the terminating NUL will be removed.

- The size in bytes of the `PACKED` array of char you use for `destination` must be exactly equal to the length of the C-style string constant including the terminating NUL, not shorter and not longer. If

  ```
  const CStringConstant = "hello"
  ```

  and you will assign the value of `CStringConstant` to `destination`, then

```
var
  destination: PACKED array[6] of char
```

Sets for 68K

This section documents anything distinctive about how sets are implemented for 68K in CodeWarrior Pascal.

Isolating one member of a set for 68K

An individual member of a set declared in CodeWarrior Pascal can be isolated using the following formula:

```
bit : BAND(SET[|SET| - 1 - EL div 8],(BSL(1,EL mod 8))
```

Enumerations for 68K

This section documents anything distinctive about how enumerations are implemented for 68K in CodeWarrior Pascal.

Storage of enumerated constants for 68K

Enumerated constants are each allocated one or two bytes of storage depending on the number of constants in the enumeration:

- One byte each if the enumeration contains 256 constants or fewer
- Two bytes each if the enumeration contains more than 256 constants

Records for 68K

This section documents anything distinctive about how records are implemented for 68K in CodeWarrior Pascal.

Record field alignment for 68K

Record fields are aligned on boundaries of:

- 2 bytes if **Struct Alignment** is set to **68K** in your 68K project

- 4 bytes if **Struct Alignment** is set to **68K 4-byte** or it's set to PowerPC and the **IEEE doubles** option is unselected.
- 8 bytes if **Struct Alignment** is set to **PowerPC** and the **IEEE doubles** option is selected.

Calling Conventions for 68K

This section describes the treatment of parameters, function results, and register conventions for the CodeWarrior Pascal compiler for 68K.

The rest of this section covers the following topics:

- Simple Type Parameters for 68K
- Structured Type Parameters for 68K
- Function Results for 68K
- A Note About 68K Registers and Mac OS

Simple Type Parameters for 68K

Parameters are evaluated from left to right and are pushed onto the stack in that order as they are evaluated. The called procedure is responsible for removing the parameters from the stack. All VAR parameters are passed as pointers to the actual storage location. Note that in cases of byte-wide parameters, the pointer may have an odd value. The Motorola 68000 processor can't address odd-numbered addresses for word and long data; the 68020, 68030, and 680040 processors can.

Non-VAR parameters are passed in the following ways, depending upon the type of the parameter:

- Values of type BOOLEAN, elements of an enumerated type with fewer than 128 elements, and subranges within the range –128..127 are passed as signed byte values. (They are pushed as bytes; the 68000 allocates two bytes for each byte on the stack.) The called procedure expects BOOLEAN parameters to be in the range 0..1.

- Values of types `INTEGER`, `UNSIGNEDWORD`, `CHAR`, and all other enumerations and subranges are passed as signed word values. Pascal `CHAR` values are expected to be 16 bit values.
- Pointers, `LONGINT`, and `UNSIGNEDLONG` values are passed as signed 32-bit values.
- Values of types `REAL`, and `DOUBLE` are passed as pointers to `EXTENDED` values. The compiler does this in a reentrant way by allocating a temporary location in the caller's activation record, converting the parameter value to an `extended` value in this location and passing a pointer to this location. The called procedure then allocates a local location of the declared type and converts the `extended` value, using the pointer, into the location and type.

Structured Type Parameters for 68K

Arrays, strings, records and sets whose size is less than or equal to four bytes are passed by pushing their value (either a 16 or 32 bit value) onto the stack. Larger arrays, strings, and records (as well as `EXTENDED` values, as mentioned above) are passed as a pointer to the value; for reentry purposes, the compiler emits code in the called procedure to copy the value to a local storage location.

Table 16.2 Parameter passing conventions for 68K

Parameter type	Pascal caller	Pascal receiver
`BOOLEAN`	Pushes byte: range 0..1	Accesses byte: range 0..1
enumeration: range 0..127	Pushes byte: range 0..127	Accesses byte: range 0..127
enumeration: range 0..255	Pushes 16 bit word: range 0..255	Accesses 16 bit word: range 0..255
enumeration: range 0..32767	Pushes 16 bit: range 0..32767	Accesses 16 bit word: range 0..32767
`CHAR`	Pushes 16 bit word: range 0..255	Accesses 16 bit word: range 0..255

subrange: range –128..127	Pushes byte: range –128..127	Accesses byte: range –128..127
subrange: range –32768..32767	Pushes 16 bit word: range –32768..32767	Accesses 16 bit word: range –32768..32767
`INTEGER`	Pushes 16 bit word: range –32768..32767	Accesses 16 bit word: range –32768..32767
`LONGINT`	Pushes 32 bit long	Accesses 32 bit signed long value
pointer	Pushes 32 bit long	Accesses 32 bit long
`REAL`	Converts to `EXTENDED`, pushes address of `EXTENDED`	Converts `EXTENDED` on stack to local `REAL`, accesses local value
`DOUBLE`	Converts to `EXTENDED`, pushes address of `EXTENDED`	Converts `EXTENDED` on stack to local `DOUBLE`, accesses local value
`EXTENDED`	Pushes address of `extended`	Copies `EXTENDED` to local `EXTENDED`, accesses local value
`ARRAY, SET, RECORD, STRING` ≤ four bytes	Pushes value (word or long)	Accesses value (word or long)
`ARRAY, SET, RECORD, STRING` > four bytes	Pushes address of value	Copies value to local, accesses local

Function Results for 68K

Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the pa-

rameters are pushed. The caller is responsible for removing the result from the stack after the call.

- For types `BOOLEAN`, `CHAR`, and `INTEGER`, `UNSIGNEDWORD`, and enumerated and subrange types, the caller allocates a 16 bit word on the stack to make space for the function result. Values of type `BOOLEAN`, enumerated types with fewer than 128 elements, and subranges within the range –128..127 are returned as signed byte values. The value goes in the low-address byte, which is the most significant byte of the word. The calling procedure expects `BOOLEAN` results to be in the range 0..1.
- `INTEGER` and `CHAR` values and all enumerated and subrange types not covered above are returned as signed word values. Pascal `CHAR` values are expected to be in the range 0..255.
- For pointers, `LONGINT`, `UNSIGNEDLONG`, and the `REAL` types, the caller allocates a 32 bit long on the stack to make space for the function result. Pointers and `LONGINT` values are returned as signed 32-bit values.
- Values of type `REAL` are returned as 32-bit real values.
- For `DOUBLE`, and `EXTENDED` types, and for sets, arrays, strings, and records greater than four bytes in size, the caller pushes a pointer to a temporary location.
- For one-byte sets and for arrays, strings, and records whose size is one word, the caller allocates a word on the stack.
- For sets, arrays, strings, and records whose size is two words, the caller allocates a long word on the stack. One-byte sets are returned as a byte value. Sets, arrays, strings, and records whose sizes are one or two words are returned as either a word or a long word.

Table 16.3 Function result passing conventions

Result type	Pascal caller	Pascal receiver	After the call
`BOOLEAN`	Allocates 16 bit word	Returns byte value: range 0..1	Pops byte

enumeration: range 0..127	Allocates 16 bit word	Returns byte value: Pops byte range 0..127	Pops byte
enumeration: range 0..32767	Allocates 16 bit word	Returns word value: range 0..32767	Pops word
`CHAR`	Allocates 16 bit word	Returns word value: range 0..255	Pops word
subrange: range –128..127	Allocates 16 bit word	Returns byte value: range –128..127	Pops byte
subrange: range –32768..32767	Allocates 16 bit word	Returns word value: range –32768..32767	Pops word
`INTEGER`	Allocates 16 bit word	Returns word value: range –32768..32767	Pops word
`UNSIGNEDWORD`	Allocates 16 bit word	Returns word value: range 0 .. 65535	Pops word
`LONGINT`	Allocates 32 bit long word	Returns long word value: range—signed 32 bits	Pops long word
`UNSIGNEDLONG`	Allocates 32 bit long word	Returns long word value: range—unsigned 32 bits	Pops long word
`REAL`	Allocates 32 bit long word	Returns `real` value	Pops real value
`DOUBLE`	Pushes address of `DOUBLE` temporary	Puts `DOUBLE` result in temporary	Pops temporary address, accesses temporary value
`EXTENDED`	Pushes address of `EXTENDED` temporary	Puts `EXTENDED` result in temporary	Pops temporary address, accesses temporary value
`ARRAY, STRING, RECORD` ≤ four bytes	Allocates 16 bit word or 32 bit long word	Returns 16 bit word or 32 bit long word	Pops 16 bit word or 32 bit long word

`ARRAY`, `STRING`, `RECORD` > four bytes	Pushes address of temporary	Puts result in temporary	Pops temporary address, accesses temporary value
`SET`: one byte	Allocates 16 bit word	Returns byte value of result	Pops byte
`SET`: one word	Allocates 16 bit word	Returns 16 bit word value of result	Pops 16 bit word
`SET`: two words	Allocates 2 16 bit words	Returns 32 bit long word	Pops 32 bit long word
`SET` > two words	Pushes address of temporary	Puts result in temporary	Pops temporary address, accesses temporary value

NOTE: Pascal does not assume any initial value for memory space allocated to a function result unless it is a pointer to a type that occupies more than four bytes of memory.

A Note About 68K Registers and Mac OS

Programming practices that have become standard for the Mac OS dictate that:

- Registers `D0`, `D1`, `D2`, `A0`, and `A1` are considered scratch registers and are not preserved across procedure calls. All other registers are preserved by the called routine.
- Register `A5` is the global data pointer.
- Register `A6` the local frame pointer.

17

Inline 68K Assembler

This chapter describes support for inline assembly language programming built into the CodeWarrior compilers.

This chapter does *not* discuss the stand-alone assembler available for 68K. For information on the stand-alone assembler, see the *Assembler Guide*.

This chapter does not document all the instructions available in 68K assembly language. For information on 68K assembly language instructions, see *M68000PM/AD: M68000 Family Programmer's Reference Manual* published by Motorola.

The sections in this chapter are:

- Working With Inline Assembly for 68K
- Assembler Directives for 68K
- Assembler Instructions for 68K

Working With Inline Assembly for 68K

This section describes how to use the CodeWarrior compiler's built-in support for assembly language programming, including assembler syntax.

The topics in this section are:

- Inline Assembler Syntax for 68K
- Defining an Assembly Function for 68K
- Creating Labels for 68K Assembly
- Using Comments in 68K Assembly

- Using the Preprocessor in 68K Assembly
- Using Structures in 68K Assembly
- Using Global Variables in 68K Assembly
- Using Local Variables and Arguments in 68K Assembly
- Returning From a Routine in 68K Assembly

Inline Assembler Syntax for 68K

To specify that a block of code in your file should be interpreted as assembly language, use the `asm` keyword.

To ensure that the C/C++ compiler recognizes the `asm` keyword, you must turn off the **ANSI Keywords Only** option in the C/C++ Language panel. This panel and its options are fully described in the *C Compilers Reference*.

The assembly instructions are the standard 68K instruction mnemonics. For information on 68K assembly language instructions, see *M68000PM/AD: M68000 Family Programmer's Reference Manual*

You use the 68K inline assembler to specify that an *entire function* is in assembly language. Assembly statement blocks within a function are not supported. In other words, the 68K inline assembler is a function-level assembler.

Inline assembly code for 68K uses the following syntax:

```
asm long f(void) { . . . } // OK: An assembly function
```

For example,

```
asm void g(void)
{
    add.l      d4, d5
    rts
}
```

However, the following code would not be legal:

```
long MyFunc (void)
{
  asm {. . .}  // Error, assembly statement blocks not supported
}
```

NOTE: Assembly language functions are never optimized, regardless of compiler settings.

Defining an Assembly Function for 68K

Keep these tips in mind as you write assembly functions:

- All statements must follow this syntax:

```
[LocalLabel:] (instruction | directive) [operands]
```

- Each instruction must end with a newline or a semicolon (;).
- Hex constants must be in C-style, not Pascal-style. For example:

```
    move.l   0xABCDEF, d5    // OK
    move.l   $ABCDEF, d5     // ERROR
```

- Assembler directives, instructions, and registers are not case-sensitive. For example these two statements are same:

```
    move.l   b, DO           // OK
    MOVE.L   b, d0           // ALSO OK
```

Listing 17.1 shows an example of an assembly function.

Listing 17.1 Creating an assembly function

```
long int b;
struct mystruct {
  long int a;
} ;
```

```
static asm long f(void)     // Legal asm qualifier
{
  move.l   struct(mystruct.a)(A0),D0 // Accessing a struct.
  add.l    b,D0   // Using a global variable, put return value
                  // in D0.
  rts             // Return from the function:
                  // result = mystruct.a + b
}
```

The rest of this section describes how to create local variables, access function parameters, refer to fields within a structure, and use the preprocessor with the assembler. A section at the end of the chapter describes some special assembler directives that the built-in assembler allows.

Creating Labels for 68K Assembly

A label must end in a colon and may contain the `@` character. For example:

```
asm void foo(void)
{
x1:  dc.b     "Hello world!\n"  // OK
@x2: dc.w     5                 // OK
x3   dc.w     1,2,3,4           // ERROR: Needs a colon
}
```

Using Comments in 68K Assembly

You cannot begin comments with a semicolon (`;`), but you can use C and C++ comments. For example:

```
    add.l      d5,d5              ; ERROR
    add.l      d5,d5              // OK
    add.l      d5,d5              /* OK */
```

Using the Preprocessor in 68K Assembly

You can use all preprocessor features, such as comments and macros, in the assembler. Just keep these points in mind when writing a macro definition:

- End each assembly statement with a semicolon (`;`), because the preprocessor ignores newlines. For example:

```
#define MODULO(x,y,result)\
          move.w   x,D0; \
          ext.l    D0; \
          divs.w   y,D0; \
          swap     D0; \
          move.w   D0,result
```

- Use `%` instead of #, since the preprocessor uses # as an operator to concatenate token. For example:

```
#define ClearD0  moveq %0,D0
```

Using Structures in 68K Assembly

You can refer to a field in a structure with the `struct` construct, as shown below:

```
struct(structTypeName.fieldName) structAddress
```

This instruction moves into `D0` the `refCon` field in the `WindowRecord` that `A0` points to:

```
  move.l  struct(WindowRecord.refCon) (A0), D0
```

Using Global Variables in 68K Assembly

To refer to a global variable, just use its name, as shown below:

```
int x;
asm void f(void)
```

```
{
  move.w      x,d0    // Moving x into d0
  // . . .
}
```

Using Local Variables and Arguments in 68K Assembly

The built-in assembler gives you two ways to refer to local variables and function arguments: you can do the work on your own or let the built-in assembler do the work for you. To do it on your own, you must explicitly save and restore processor registers and local variables when entering and leaving your assembly function. You cannot refer to the variables by name. You can refer to function arguments off the stack pointer. For example, this function moves its argument into `d0`:

```
asm void foo(short n)
{
  move.w      4(sp),d0 //  n
  // . . .
}
```

To let the built-in assembler do it for you, use the directives `fralloc` and `frfree`. Just declare your variables as you would in a normal C function. Then use the `fralloc` directive. It makes space on the stack for the local stack variables and reserves registers for the local register variables (with the statement `link #`*x*`,a6`). In your assembly, you can refer to the local variables and variable arguments by name. Finally, use the `frfree` directive to free the stack storage and restore the reserved registers.

Listing 17.2 is an example of using local variables and function arguments.

Listing 17.2 Using the fralloc directive

```
static asm short f(short n)
{
```

```
  register short a; // Declaring a as a register variable
  short b;          // and b as a stack variable
  // Note that you need semicolons after these statements.

  fralloc +     // Allocate space on stack and reserve registers.
  move.w  n,a   // Using an argument and local var.
  add.w   a,a
  move.w  a,D0

  frfree        // Free the space that fralloc allocated
  rts
}
```

Returning From a Routine in 68K Assembly

Every assembly function should end in an `rts` or a `preturn` statement. If you forget to add one, the compiler does not add one for you, and does not raise an error. Use the `rts` statement for ordinary C functions. Use the `preturn` statement for Pascal functions, since it performs the clean up that Pascal functions need. For example:

```
asm void f(void)
{
    add.l      d4, d5
}                       // Error, no RTS statement

asm void g(void)
{
    add.l      d4, d5
    rts                 // OK
}

asm void pascal h(void)
{
    add.l      d4, d5
    preturn             // OK
}
```

Assembler Directives for 68K

This section describes some special assembler directives that the 68K built-in assembler accepts. They are:

- dc
- ds
- entry
- fralloc
- frfree
- machine
- opword

dc

`dc[.(b|w|l)]` *constexpr* `(,`*constexpr*`)*`

68K assembler directive that defines a block of constant expressions, *constexpr*, as initialized bytes, words, or long words. If there is no qualifier, `.w` is assumed. For `dc.b` you can specify any string constant (C or Pascal). For `dc.w` you can specify any 16-bit relative offset to a local label. For example:

```
asm void foo(void)
{
x1: dc.b  "Hello world!\n" // Creating a string
x2: dc.w  1,2,3,4          // Creating an arrray
x3: dc.l  3000000000       // Creating a number
}
```

ds

`ds[.(b|w|l)]` *size*

68K assembler directive that defines a block of *size* bytes, words, or longs. The block is initialized with null characters. If there is no qualifier, `.w` is assumed. For example, this statement defines a block big enough for the structure `DRVRHeader`.

```
    ds.b  sizeof(DRVRHeader)
```

entry

```
entry [extern|static] name
```

68K assembler directive that defines an entry point into the current function. Use the `extern` qualifier to declare a global entry point and use the `static` qualifier to declare a local entry point. If you leave out the qualifier, `extern` is assumed.

Listing 17.3 Using the entry directive

```
static long MyEntry(void);
static asm long MyFunc(void)
{
    move.l  a,d0
    bra.s   L1

    entry   static MyEntry
    move.l  b,d0
L1: rts
}
```

fralloc

```
fralloc [+]
```

68K assembler directive that lets you declare local variables in an assembly function. The `fralloc` directive makes space on the stack for your local stack variables and reserves registers for your local register variables (with the statement `link #x,a6`). For more information, see "Using Local Variables and Arguments in 68K Assembly" on page 344.

There are two versions of `fralloc`. The `fralloc` directive (without a `+`), pushes modified registers onto the stack. The `fralloc +` directive also pushes all register arguments into their 68K registers.

frfree

```
frfree
```

68K assembler directive that frees the stack storage area and restores the registers (with the statement `unlk a6`) that `fralloc` reserved. For more information, see "Using Local Variables and Arguments in 68K Assembly" on page 344.

machine

```
machine number
```

68K assembler directive that specifies which CPU the assembly code is for. The *number* must be one of the following:

68000	68010	68020	68030
68040	68349	68881	68882
68851			

To use the following MC68020 assembler instructions, specify `68020`, `68030`, or `68040`:

bfchg	bfclr	bfexts	bfextu
bfffo	bfins	bfset	bftst
divsl	divs.l	divul	divu.l
muls.l	mulu.l	extb.l	rtd

You cannot use MC68020, MC68030, or MC68040 addressing modes. To disable the MC68020 assembler instructions, specify `68000` or `68010`. The arguments `68349`, `68881`, `68882`, and `68851` have no effect.

opword

`opword` *const-expr* `(,`*const-expr*`)*`

68K assembler directive that lets you include the opcode for an instruction. It works the same as `dc.w`, but emphasizes that the expression is an instruction. For example, this directive calls `WaitNextEvent()`:

```
opword  0xA860      // WaitNextEvent
```

Assembler Instructions for 68K

The built-in assembler uses all the standard MC 68000 assembler instructions. It accepts some additional directives described in "Assembler Directives for 68K" on page 346. It also accepts the following 68020 assembler instructions, after you use one of these directives: `machine 68020`, `machine 68030`, or `machine 68040`:

`bfchg`	`bfclr`	`bfexts`	`bfextu`
`bfffo`	`bfins`	`bfset`	`bftst`
`divsl`	`divs.l`	`divul`	`divu.l`
`muls.l`	`mulu.l`	`extb.l`	`rtd`

You cannot use MC68020, MC68030, or MC68040 addressing modes.

TIP: If you know the opcode for an assembly statement that's not supported, you can include it in your function with the `opword` directive, described at "opword" on page 349.

18

Debugger Setup

This chapter explains how to configure the IDE debugger to effectively debug projects on the Mac OS, including how to debug code running on a Windows machine while using the Macintosh-hosted IDE debugger.

There are three primary topics discussed in this chapter. They are:

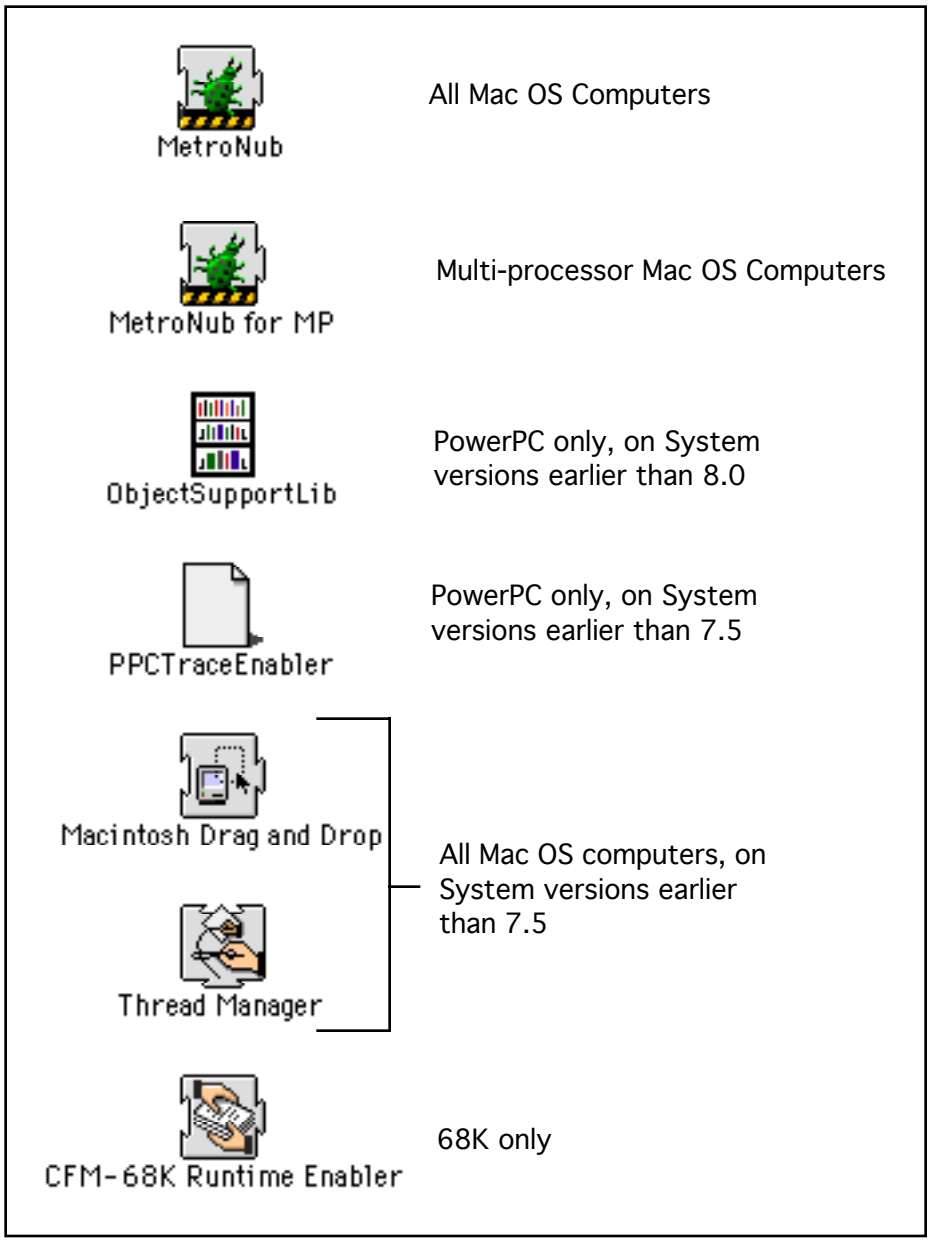
- Debugger Components
- Configuring the Cross-Debugger
- Debugger Preferences

See also *Debugger User Guide* for more detailed information on the IDE debugger.

Debugger Components

In order to work, the debugger also requires a few system extensions to be installed in the `Extensions` subfolder of your `System` folder. The installer will automatically place these items in your Extensions folder; if you install or remove any of them, you must restart your computer for the changes to take effect. Before using the debugger, make sure that the correct extensions are installed for your current target and platform, as shown in Figure 18.1.

Figure 18.1 Extra extensions required by the IDE debugger



- MetroNub—CodeWarrior's own *debugger nub*: a system extension that provides low-level services so that the IDE debugger can control your program. It allows you to debug any kind of object code created by the CodeWarrior environment.

- ObjectSupportLib—a shared library needed for debugging PowerPC object code on System versions earlier than 8.0.
- PPCTraceEnabler—needed for debugging PowerPC object code on System versions earlier than 7.5.
- CFM-68K Runtime Enabler—required to use the debugger shared library. You can find a version of the debugger on the Tools CD that does not use the shared library.

WARNING! Earlier versions of CodeWarrior's debugger (MW Debug) used Apple's DebuggerINIT and Power Macintosh Debug Services nubs when running under System versions earlier than 7.5. These nubs are no longer supported; you must use MetroNub for all debugging.

Configuring the Cross-Debugger

The CodeWarrior product for Macintosh-hosted development includes a facility for debugging code running on a Windows machine while using the Macintosh-hosted IDE debugger. This activity is called cross debugging.

To enable cross debugging, you must install the appropriate software and connect the two machines via a network link.

This section discusses the following topics:

- Installing Mac OS Debugger Software
- Configuring Windows/Mac Debug Connectivity
- Notes on Using the Cross Debugger

Installing Mac OS Debugger Software

This section discusses the requirements for the operating systems on your machines, and how to install the necessary software.

Operating System Requirements

In order to use the cross debugger facilities, you must have a Mac OS computer running System 7.5.3 or later, with Open Transport 1.1 or later installed. There are no RAM requirements. `MetroNub` doesn't use very much memory under the Mac OS.

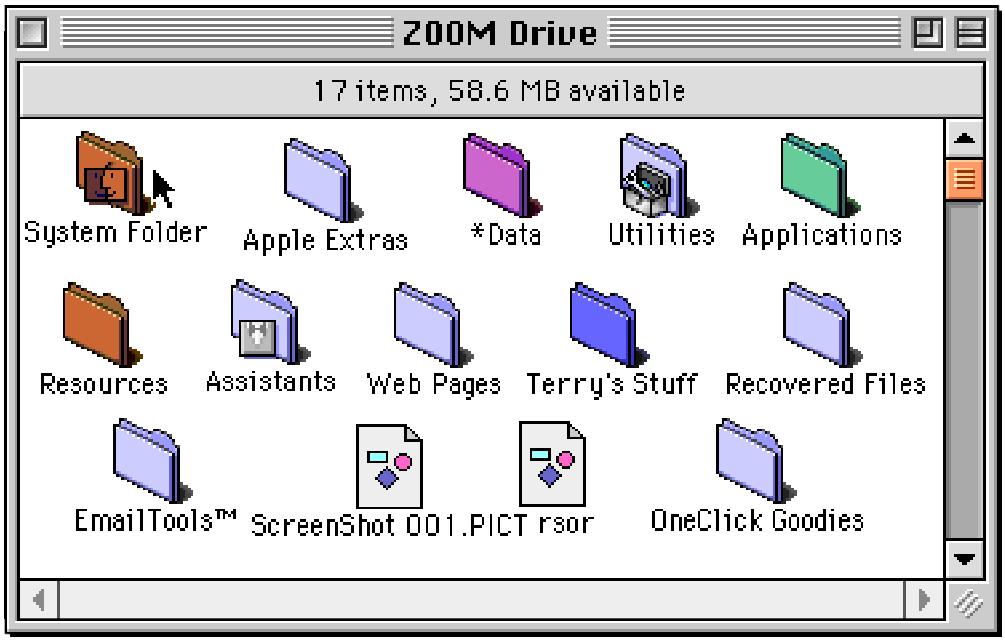
[Ken - Do we need to keep the Windows requirements?]

For Windows, you need to be running Windows 98, Windows 95, or Windows NT 4.0.

Installing the Mac OS Software

To use the debugger on Macintosh to debug code running on Windows, you will need to first install the `MetroNub` Extension onto the Mac OS computer from which you will be debugging the code. To do this, locate the file called `MetroNub` on your CodeWarrior CD and drag it onto the `System Folder` on your Mac OS machine. The Mac OS will put the file in the right place inside the `System Folder`.

Figure 18.2 Mac OS System Folder



After installing the software, restart your Mac OS computer so the software will be loaded at boot time. Until you restart your Mac OS computer, you will not be able to use the cross debugging facilities.

The `Control Panels` folder is inside the `System Folder` on your Mac OS machine, and so is the `Extensions` folder. The active `System Folder` has a special icon on it. Refer to Figure 18.2, and notice the special icon on the `System Folder` in the upper-left portion of the figure. To see the `Control Panels` and `Extensions` folders, double-click on the `System Folder`.

Configuring Windows/Mac Debug Connectivity

This section discusses the hardware and software configuration necessary for enabling cross debugging.

Network Connection (Hardware)

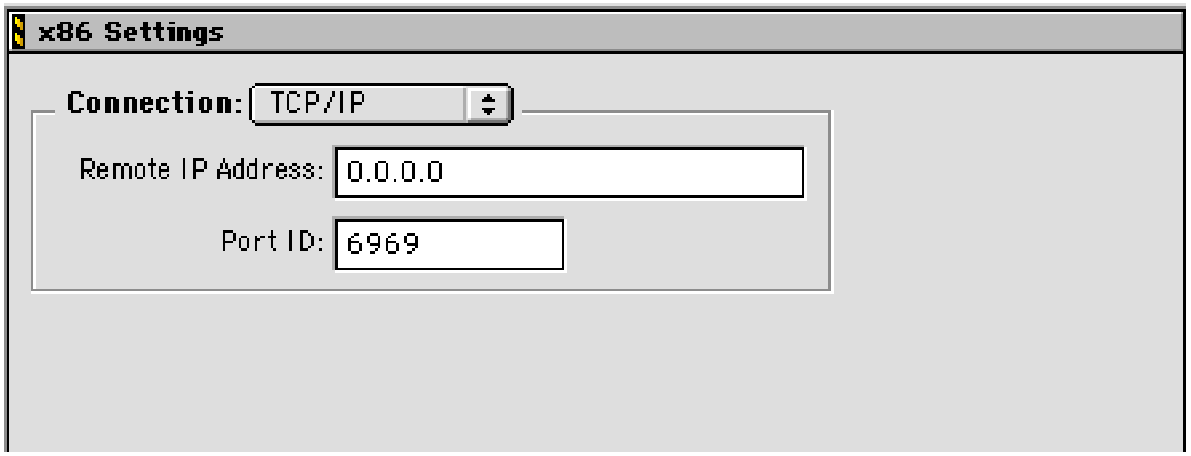
The two machines will communicate with each other to control debugger operations using the TCP protocol. You should make sure that you have a reliable TCP/IP connection between the Mac OS computer and the Windows computer. Refer to the documentation that came with your networking hardware for more information on how to configure it.

Network Connection (Software)

The **x86 Settings** preference panel allows connections to your Windows machine from your Macintosh. Make sure to fill in the **Remote IP Address** field in the **x86 Settings** preference panel, as shown in Figure 18.3. This should be the IP address of your Mac OS computer. You can find the IP address of your Mac OS computer in the MacTCP or TCP/IP Control Panels. If you can't find it there, ask your system administrator.

NOTE: Do not change the **Port ID** field in the **x86 Settings** panel. This number is fixed.

Figure 18.3 x86 Settings preference panel



Notes on Using the Cross Debugger

[Ken - does this section seem necessary? It seems to have a hodge-podge of stuff and not very much meat to it. Let me know what you think. Is there more info that should be included to make it better?]

The **Stop** and **Kill** commands might not always work if the program you are debugging is in an infinite loop. If this is the case, you will need to reset the target machine.

For x86 debugging, the sym file name must end with `.iSYM`. Sym files ending with `.SYM` are assumed to be 68K and `.xSYM` is assumed to be PowerPC.

To learn more about how to use the debugger, refer to the *Debugger User Guide* on your CodeWarrior CD.

Debugger Preferences

This section discusses preference settings for the debugger that are specific to the Mac OS. Topics discussed in this section include:

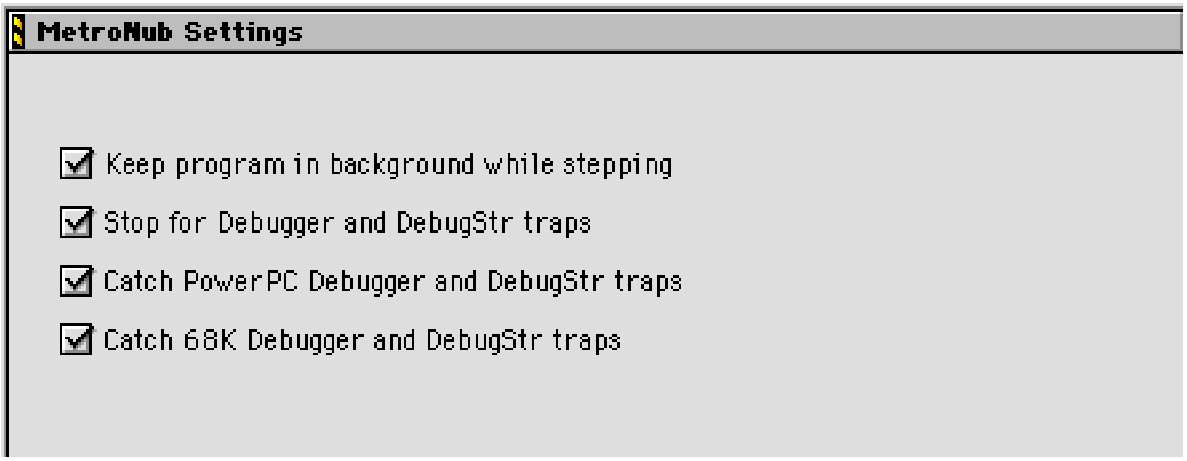
- MetroNub Settings

MetroNub Settings

Figure 18.4 shows the MetroNub Settings panel. This preference panel has four checkbox options:

- Keep program in background while stepping
- Stop for Debugger and DebugStr traps
- Catch 68K Debugger and DebugStr traps
- Catch PowerPC Debugger and DebugStr traps

Figure 18.4 MetroNub Settings preference panel


MetroNub Settings
Keep program in background while stepping
Stop for Debugger and DebugStr traps
Catch PowerPC Debugger and DebugStr traps
Catch 68K Debugger and DebugStr traps

Keep program in background while stepping

Allows you to step through a program's code while the program is running in the background, without bringing it to the foreground at each step. This makes stepping faster and avoids screen flicker resulting from the program's windows flashing to the front and then back again each time you step.

WARNING! This option may affect the execution of your program if the program behaves differently in the background than in the foreground.

Stop for Debugger and DebugStr traps

Stops execution of the target program whenever it executes a `DebugStr()` or `SysBreakStr()` call. If this preference is deselected, the debug string is logged without stopping program execution.

See also "Stopping Under Program Control" on page 359 for information on `Debugger()`, `DebugStr()`, and `SysBreakStr()`.

Catch 68K Debugger and DebugStr traps

Configures the debugger to catch the programmed debugger breaks `Debugger()` and `DebugStr()` when issued from 68K code. If this preference is deselected, these breaks will be passed through to the low-level debugger.

Catch PowerPC Debugger and DebugStr traps

Configures the debugger to catch the programmed debugger breaks `Debugger()` and `DebugStr()` when issued from PowerPC code. If this preference is deselected, these breaks will be passed through to the low-level debugger.

19

Debugging for Mac OS

This section provides information on how to debug special types of Mac OS code.

To learn more about how to configure the CodeWarrior IDE for debugging projects, refer to "PPC Linker" on page 195 and "68K Linker" on page 291.

The topics discussed in this chapter are:

- Debugging Concepts
- Debugging Shared Libraries (PowerPC)
- Debugging Code Resources

Debugging Concepts

This sections introduces you to one of the main concepts when debugging source code:

- Stopping Under Program Control

Stopping Under Program Control

Besides setting manual breakpoints from the debugger, you can also embed breakpoints directly in your source code with Mac OS Toolbox calls. A program can stop itself and turn over control to the debugger by calling `Debugger()`, `DebugStr()`, `SysBreak()`, or `SysBreakStr()`.

You must also have a low-level debugger, such as MacsBug, installed on your machine *in addition to* the CodeWarrior debugger.

WARNING! *If a low-level debugger is not installed, these calls may crash your machine.* This is true even if the CodeWarrior debugger is running.

Both `SysBreak()` and `SysBreakStr()` stop the program and activate the CodeWarrior debugger (provided that it is already running), just as if it had encountered a manually-set breakpoint. You can then examine the call chain, inspect or change variables, step through your code, or use any of the IDE debugger's other facilities. The difference between the two calls is that `SysBreakStr()` takes a Pascal-style string as a parameter and displays it in the log window when the debugger is activated, allowing you to send informational messages to yourself while debugging.

The `Debugger()` and `DebugStr()` calls are analogous to `SysBreak()` and `SysBreakStr()`, except that you can control whether they break to the CodeWarrior debugger or to your low-level debugger. If either of these calls is issued from 68K code and the **Catch 68K Debugger DebugStr traps** preference is set in the **MetroNub Settings** panel, the IDE debugger gets control; if this setting is turned off, control goes to the low-level debugger instead. The calls work the same way when issued from PowerPC code, except that they are governed by the **Catch PowerPC Debugger and DebugStr traps** setting instead.

Table 19.1 summarizes the behavior of these four system calls. If the CodeWarrior debugger is not running, any of the four calls will drop you into the low-level debugger.

Table 19.1 Calling debugger at runtime

Debugger activated:	Without a debug string	With a debug string
Always CodeWarrior	`SysBreak()`	`SysBreakStr()`
CodeWarrior or low-level, according to applicable preference setting	`Debugger()`	`DebugStr()`

Another setting in the **MetroNub Settings** panel, **Stop for Debugger and DebugStr traps**, controls whether the debugger stops the program or lets it continue on receiving a `DebugStr()` or `SysBreakStr()` message. When this preference is turned off, such messages are simply sent to the log window without stopping program execution. You can use this feature to keep a running record of important events as they occur in your program. If you keep the log window visible on your screen, you can watch the messages go by "on the fly" as your program runs.

NOTE: Strings ending with the characters `;g` or `;G` are always logged without stopping program execution, regardless of the setting of the **Stop for Debugger and DebugStr traps** preference. This feature is compatible with both MacsBug and MacApp.

See also "MetroNub Settings" on page 357 and the *Debugger User Guide* for more details on the Log Window and debugger preferences.

Debugging Shared Libraries (PowerPC)

You can use the debugger to debug dynamically loaded, disk-based shared libraries—those loaded with the `GetDiskFragment()` and `GetSharedLibrary()` operating-system calls. You can also debug shared libraries when no SYM file is available for the application, as is the case with shared library plug-ins.

The topics discussed in this section are:

- Shared-Library Limitations
- Debugging a Shared Library in Four Steps

Shared-Library Limitations

There are three limitations on debugging shared libraries:

- The debugger does not support code loaded via `GetMemFragment()`.

- The debugger doesn't support multiple instances of the same library in an application context. It may get confused about which data section is current and show the wrong global data for the library.
- The CFM (Code Fragment Manager) fragment name must match the shared library's file name unless you enter a fragment name in the CFM68K or PPC PEF Target Settings panel or create your own `'cfrg'` resource.

Debugging a Shared Library in Four Steps

1. **Make sure the shared library file has a matching SYM file.**

   As with applications, a shared library's SYM file must have the same name as the shared library, plus the PowerPC extension `.xSYM`. For example, the file could be name MyShlib.xSYM.

2. **Enable the IDE debugger and open the shared library's `.xSYM` file.**

3. **Set a breakpoint somewhere in the shared library's source or assembly code.**

4. **Launch the application that calls the shared library.**

   If you have a `.xSYM` file for the application, open that file and the application will be launched in the usual way.

   If you do not have a `.xSYM` file, launch the application in the Finder while holding down the **Control** key. The **Control** key notifies the debugger when the application is loaded; you can release it as soon as you see the debugger come to the foreground to open a process window for the new application.

   It is not strictly necessary to open the `.xSYM` files for all your shared libraries before launching the application. If you open the `.xSYM` file for a shared library that is in use by an application already being debugged, the debugger will connect the `.xSYM` file to the shared library and allow it to be debugged.

Debugging Code Resources

There are many different types of code resources. The CodeWarrior debugger can debug many of them, but not all. Examples of types for which the debugger is known to work include PhotoShop™ plug-ins, Quark™ XTensions, `'XCMD'` resources, and 4D™ extensions.

This section discusses the following topics:

- Debugging Requirements
- Runtime Debugging Requirements
- Debugging a Code Resource in Four Steps

Debugging Requirements

There are some requirements you must meet if you wish to debug a code resource.

- Make sure you generate a SYM file when you build the project.
- Make sure the resource types and IDs at debug time are the same as when the project was linked.

If you meet these requirements, the code resource can be single- or multi-segment. If it is a 68K project, you can use A4 or A5 globals, depending upon the needs of your project.

Runtime Debugging Requirements

At runtime, the code resource being debugged must meet certain conditions:

- Make sure the code resource isn't detached from its resource file or moved to some other part of memory before execution. If the resource moves after the debugger has located it in memory, the debugger will fail.
- If you are using the A4 register to reference global variables in 68K code, set A4 with the `SetCurrentA4()` function. Otherwise the debugger may be unable to display your global variables, or may display them with incorrect values.

- The debugger cannot debug interrupt-level code or code that might cause Operating System reentrancy issues. Because it is a high-level debugger, the IDE debugger shares the operating system with all other applications. For example, imagine trying to debug an `'MDEF'`—a menu definition function. If you stop at a breakpoint in the `'MDEF'`, the Menu Manager is in an undefined state because it was in the middle of carrying out a request when the breakpoint was hit. If you then do anything while stopped that uses the Menu Manager, the operating system may crash.

Debugging a Code Resource in Four Steps

1. **Make sure the debugger properly matches the SYM file to the code resource.**

   For example, a project creates a code resource, places it in a file named `MyCodeRes`, and generates a corresponding SYM file. The debugger targets the file `MyCodeRes` as the executable file. If the debugger can't find `MyCodeRes`, it will prompt you to locate the executable file. Select the file containing the code resource, which may be in its own file or in another file such as an application.

2. **Enable the IDE debugger and open the code resource's SYM file.**

3. **Set a breakpoint in the code resource's source or assembly code.**

   Ideally, place a breakpoint at the code resource's entry point, so that the debugger will get control as soon as the code resource is called.

4. **Run the program that calls the code resource.**

   Launch the host program that calls the code resource as you would normally. When the host program calls your code resource, the debugger takes control. You can then debug the code resource. For example, to debug a typical application plug-in, launch the application and issue the command that calls your plug-in.

TIP: You can also have the IDE launch the application for you. See "Debugger Settings" on page 204 for more info

---

TIP: If you are debugging a code resource as well as its host program, you can have browser windows for both. With multiple SYM files open, you can debug your application and the code resource simultaneously, or even debug multiple code resources.

20

Advanced Debugging

There are many special types of code that can be used on Mac OS computers. Each type has its own idiosyncrasies for writing and debugging.

To learn more about how to configure the CodeWarrior IDE for debugging projects, refer to "PPC Linker" on page 195 and "68K Linker" on page 291.

This chapter discusses the procedures for:

- Debugging C++ and PowerPlant
- Debugging SOM
- Debugging Threads
- Debugging Profiled Code

Debugging C++ and PowerPlant

In this section we discuss various aspects of debugging C++ and PowerPlant code. The topics in this section include:

- Object-Oriented Data Types and References
- Debugging Objects Created With new
- Debugging Static Constructors and Destructors
- Viewing PowerPlant Source Code
- Debugging Inline Functions
- Debugging Template Classes
- Debugging C++ Exceptions

Object-Oriented Data Types and References

The debugger helps you debug C++ objects, including PowerPlant objects. You may declare in your source code that an object belongs to a certain base class, but the actual object at runtime may be a member of some subclass. The debugger dynamically recognizes the runtime class of an object and displays it with its true class, rather than as an object of the declared class.

Listing 20.1 C++ Example

```
void
LBroadcaster::BroadcastMessage(
  MessageT  inMessage,
  void     *ioParam) const
{
  if (IsBroadcasting()) {
    LListIterator iterator(mListeners,
                           iterate_FromStart);
    LListener *theListener;
    while (iterator.Next(theListener)) {
      if (theListener->IsListening()) {
        theListener->ListenToMessage(inMessage,
                                     ioParam);
      }
    }
  }
}
```

Consider the example shown in Listing 20.1, taken from the PowerPlant class `LBroadcaster`. The local variable `theListener` has a declared type of `LListener*`. If you are stepping through this routine at runtime, this object will certainly be an instance of some subclass, such as `LDialogBox`, `LRadioGroup`, or a subclass of your own. The debugger determines the object's actual class and automatically casts it to that type. You can then see all of the object's data members and easily pick out which instances you want to trace further.

If you enable the **Show Types** command in the **Data** menu of the CodeWarrior Debugger, the object's runtime type appears in the locals pane of the program window as well. C++ reference variables and parameters are displayed in the debugger as pointer types, because that is how they are stored in the SYM file.

For more information on this topic, please refer to the section entitled "Attempt to use dynamic type of C++ or Object Pascal objects" in the *Debugger User Guide*.

Debugging Objects Created With new

Objects created via the C++ `new` operator are heap-based. After allocating space in the heap for the new object, the `new` operator calls the object's constructors in a specified order. If the new object has data members that are also objects, the member objects are also created via the `new` operator. Table 20.1 lists the sequence of events that occur when initializing a heap-based object.

Table 20.1 Sequence of execution for creating heap-based objects

1. `new` operator
2. base class constructor(s)
3. member object constructor(s)
4. this class constructor

When using the library version of the `new` operator, correspondence between the SYM file and your source code is not available. If you step into the `new` operator under these circumstances, step through it in the assembler view or use the **Step Out** command. On the other hand, if the source code for the `new` operator is included in the project, you can step into and trace through the `new` operator.

NOTE: If the **Don't Step Into Runtime Support Code** preference is selected and no source code is available from the SYM file, the CodeWarrior Debugger skips these routines.

After the `new` operator completes execution, you can step into the object's constructors (steps 2, 3, and 4 in Table 20.1). You can step through all constructors, whether called explicitly or implicitly. To step through a series of constructors, use the **Step Into** command repeatedly while the current-statement arrow is at the first line of the constructor, just as if there were multiple function calls on that line.

NOTE: Synthesized constructors do not have source correspondence, because there is no source code associated with them.

Listing 20.2 shows some example code that we'll use to demonstrate how to step into a series of constructors. If you begin debugging from the statement in `main()` that allocates `b` (using the `new` operator), the following sequence occurs:

1. **Using the Step Into command goes to the `new` operator.**

   When the `new` operator returns, the current-statement arrow returns to the line in `main()` where `b` is allocated.

2. **Using Step Into again gets you into the constructor for `B`, at the first line in `B::B()`.**

3. **Continuing to use Step Into gets you into the constructor for `B`'s base class, `A::A()`.**

   When it returns, **Step Into** brings the current statement arrow to the `A` data member.

4. **When the `A` constructor member is done, you can continue using Step Into through the `B` constructor.**

Listing 20.2 Stepping through heap-based objects

```
class A
{
  public:
    A();
};

class B : public A
{
```

```
  public:
    B();

    A myA;  // Contrived example: B derives
            // from A and has a member A.
};

void main()
{
  B* b = new B;
}

A::A()
{
  ...
}

B::B()
{
  ...
}
```

Debugging Static Constructors and Destructors

Static objects are objects that are declared as global or static variables. If such an object has a constructor, the constructor is executed before entering the program's main routine. Since, by default, the debugger launches your application and stops it at the beginning of your main routine, static constructors have already run by the time you get control.

To prevent the automatic execution of static constructors, deselect the **Automatically Launch Applications When SYM File Opened** preference in the **Global Settings** panel. This allows you to set breakpoints as desired in any static constructors before launching your program using the **Debug** command.

TIP: You can also prevent static constructors from running automatically by hold down the Option key as the SYM file is being opened.

Static destructors execute after the program's main routine ends, so you can step through them by simply setting breakpoints in the normal way.

Viewing PowerPlant Source Code

When you are debugging an application written with PowerPlant, add the source code files to your project instead of the PowerPlant libraries or PowerPlant subprojects. Using the PowerPlant source code files lets you view PowerPlant methods at the source level instead of the assembler level.

When your application is ready to ship, you can speed compilation and link times, test, and ship the final version of your application with the PowerPlant libraries, or by using the PowerPlant project files as subprojects. Be sure to test your application thoroughly if you use the libraries or subprojects instead of the source code.

To learn more information about subprojects, refer to the *IDE User Guide* documentation.

Debugging Inline Functions

The CodeWarrior linker can generate SYM information for code that comes from a file other than the main source file, such as a header file. Because inline functions are typically defined in a header file, this means you can debug inline functions *when they are not actually inlined*.

To debug inline functions, go to the C/C++ Language preference panel and turn on the **Don't Inline** preference. The compiler will treat the inline function as if it were a regular function, and the linker will generate SYM information for it. Because SYM information is available, you can debug the function like any other. When

you are through debugging, you can turn the **Don't Inline** preference off so that the compiler will actually generate inline code.

Even if you designate a function as inline and have turned off the **Don't Inline** preference, the compiler may still ignore your request and not inline the function. CodeWarrior C++ inlines a function only if it satisfies *all* of the following conditions:

- It is declared with the `inline` keyword.
- It returns its result in a register.
- It has only one `return` statement.
- It has a definite number of arguments (no variable argument lists).
- If it has any class arguments, they do not have destructors.
- The function call
  - is at the statement level, or
  - is at the expression level and the function contains no control statements (such as `if`, `while`, `for`, or `switch`).

If the inline function is actually inlined, there is no source-code correspondence information for the function in the SYM file. You can step through the code in assembler view, but you cannot see the original source code that defined the function.

For more information on when the compiler does and does not inline a function, refer to the *C Compilers Reference.*

Debugging Template Classes

NOTE: Consult the release notes for the CodeWarrior compilers and linkers for any late-breaking news on this topic.

Under normal circumstances, you cannot debug the member functions of an object derived from a template class. However, you can get around this limitation if you specialize on the template class. Specializations are allowed in C++ to allow for the possibility of

doing particular optimizations in template methods, depending on the data type.

You can take advantage of this feature to specify a specialized data type based on the template class. For example, if the `short` data type makes sense, define a `short` version of your template class, such as

```
void MyTemplateClass<short>::MyFunction(short
myData)
{
  ...
}
```

When you call `MyFunction()` with a `short` parameter, the debugger can see your code and allows you to stop there. When you are finished debugging your code, you can throw away the specializations (assuming you aren't actually using them for special cases).

Debugging C++ Exceptions

The CodeWarrior Debugger provides direct support for debugging C++ exceptions. To activate this feature, choose **Break on C++ Exception** from the **Control** menu. Every time an exception is thrown, the debugger will break at the `throw()` function defined in `NMWException.cp`.

The `NMWException.cp` file does not need to be included in your project. If the file is not in your project, the debugger will prompt you to locate this source file the first time an exception is thrown. At the time of this writing, the path to the file is `Mac OS Support:Libraries:Runtime:Common Sources`. Your mileage may vary.

When you break in `throw()`, you can use the stack crawl pane of the IDE debugger to return to the precise location in the `try` block that generated the `throw`. You can step through the `throw()` code if you wish. If you want to proceed directly to the `catch` block responsible for handling this exception, you can use either the **Run** or **Step Out** command. Either one will exit from `throw()` and stop at the end of the `try` block that caused the exception. At that point, you can step through your `catch` block and watch what happens.

Debugging SOM

The CodeWarrior debugger gives you direct support for debugging System Object Model (SOM) objects. You use SOM objects when writing an OpenDoc part.

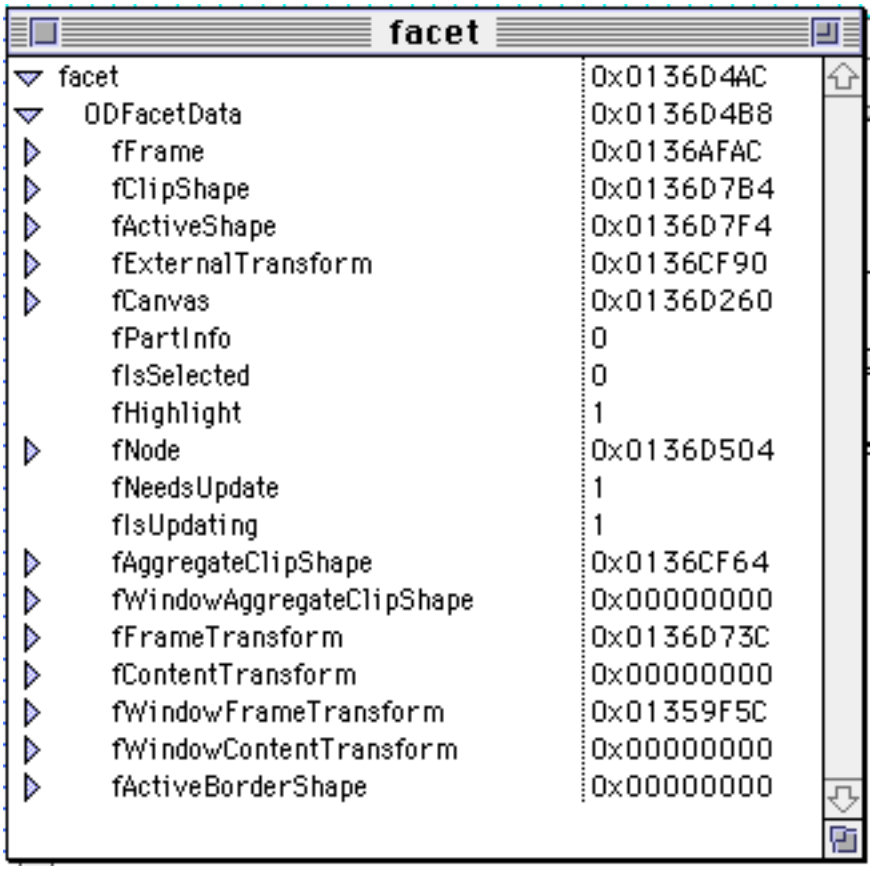
To debug code with SOM objects, select the **Attempt to use dynamic type of C++, Object Pascal and SOM objects** checkbox on the **Display Settings** preference panel in the **IDE Preferences** dialog box. To enable the **IDE Preferences** dialog, choose **Preferences** from the **Edit** menu in the IDE.

To learn more on this topic, or to learn about how to change debugger preferences, refer to the *Debugger User Guide*.

Viewing SOM Objects

You can view a SOM object in the variable pane, or you can double-click the variable to see its contents in a separate window. Figure 20.1 shows a SOM object variable expanded in its own window.

Figure 20.1 Viewing a SOM Object in its Own Window



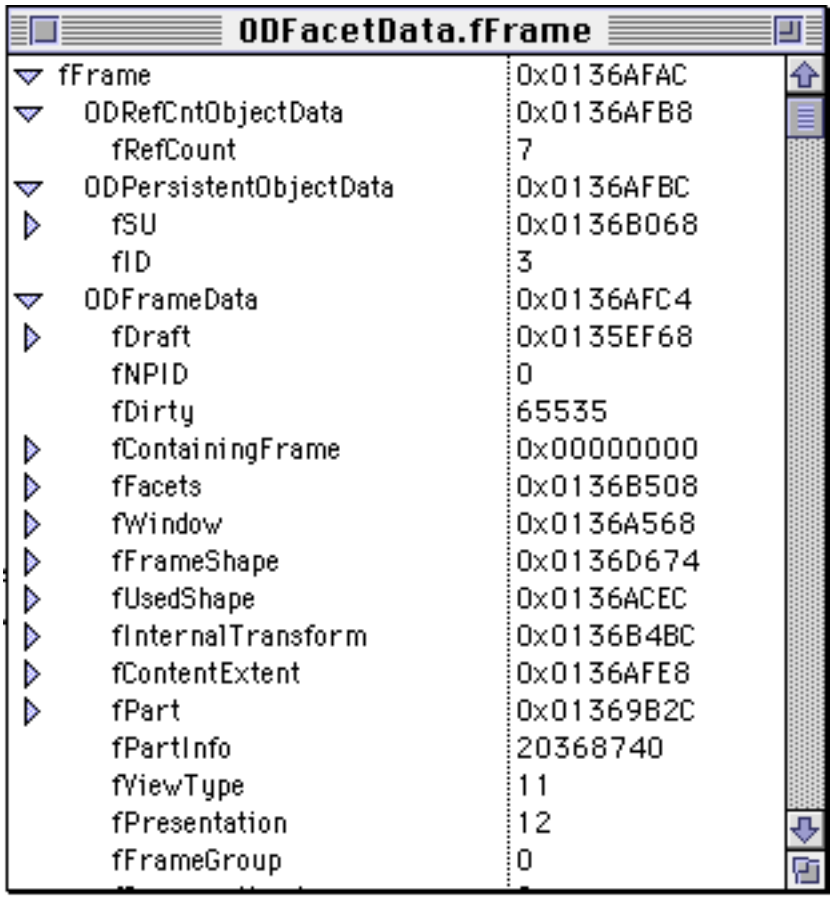
Note that the `facet` variable has an `ODFacetData` structure embedded inside. In C++, when you look at the contents of an object, you see all the data members listed as belonging directly to the object. Data members inherited from base classes are mixed in with other data members. SOM stores inherited data members from each base class in the hierarchy separately within the object. So, assume you have an `ODFrame` object that inherits from this hierarchy:

SOMObject -> ODObject -> ODRefCntObject -> ODPersistentObject -> ODFrame.

When you look at the object in the debugger, it looks like Figure 20.2. You have the data members from `ODRefCntObject`, `ODPersistentObject`, and `ODFrame` listed separately. The `SO-`

`MObject` and `ODObject` don't appear because they have no data members.

Figure 20.2 Frame variable in its own window



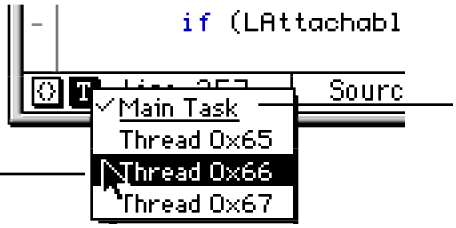
NOTE: SOM adds "`Data`" to the structure name for the inherited class it represents. So, the class `ODFrame` becomes `ODFrame-Data` in Figure 20.2.

Debugging Threads

The IDE debugger supports the Macintosh Thread Manager. The task pane of the Process window displays a list of current threads and allows you to view the call chain, variables, and source code of any thread.

The *threads pop-up menu* lets you view the call chain, variables, and source code of any thread running under the Thread Manager (Figure 20.3). All the panes in the stack crawl window update when you switch threads.

Figure 20.3 Threads pop-up menu


if (LAttachabl
Sourc
Main Task
Thread 0x65
Thread 0x66
Thread 0x67
Chooseing a thread displays all the information for the thread. A checkmark indicates that thread is currently displayed.
The underlined thread, in this case Main Task, is the thread currently executing.

The threads pop-up menu is only available if the **Show tasks in separate windows** preference has been turned off in the **Display Settings** panel.

See also The *IDE User Guide*.

NOTE: As with other low-level code, the debugger cannot debug 68K preemptive threads.

There is an important distinction to be made between cooperative threads and preemptive tasks or threads. Under the Mac OS Thread Manager, threads are usually cooperative. It is also possible to write code using preemptive threads on 68K Mac OS computers, but not on Mac OS PowerPC computers.

NOTE: Apple no longer supports preemptive threads with the Thread Manager.

When you stop in a cooperative task under the Mac OS (due to a breakpoint, step or some sort of exception) all other cooperative tasks are also stopped. Similarly, when you resume a cooperative task (step or run) all the other cooperative tasks are also resumed.

However, this is not the case of preemptive tasks (which currently only appear on multi-processor computers or when using 68K preemptive threads). Preemptive tasks can be suspended and resumed completely independently of each other and of the cooperative tasks.

To learn more information on this topic, consult the *Debugger User Guide* for more information. You may also wish to consult Apple Thread Manager SDK, available on the Apple Developer CD subscription from Apple Computer.

Debugging Profiled Code

It is possible to debug code that has calls to the profiler in it. However, the profiler does interfere with stepping through code. You may find it simpler to debug non-profiled code, and profile separately. In this section you'll learn what happens when you step into a profiled routine and step out of a profiled routine. Information about the effect that stopping in the debugger has on the profile results is also presented here. For more information on the profiler, refer to the *Profiler Manual.*

Stepping into a profiled routine

If you step into a profiled routine, you'll see assembly code instead of source code. The compiler has added calls to `__PROFILE_ENTRY` at the start of the routine. This is how the profiler knows when to start counting time for the routine.

If you step through the assembly code far enough to get to the code derived from the original source code, then switch the view from

source to assembly and back again, you can see the original source code.

Stepping out of a profiled routine

If you single-step out of a routine being profiled, you end up in the `__PROFILE_EXIT` assembly code from the profiler library. This is how the profiler knows when to stop counting time for the routine.

Effect of stopping on the profile results

If you stop in a profiled routine, the profiler counts all the time you spend in the debugger as time that routine was running. This skews the results.

WARNING! If you debug profiled code, you must be careful not to kill the code from the debugger. Remember, if you have called `ProfilerInit()` you must call `ProfilerTerm()` on exit or you may crash the computer. Before using the profiler and debugging profiled code, read the *Profiler Manual*.

21

DebugNew

DebugNew is a C++ source code library you can add to your C++ or PowerPlant projects to help you find memory problems. DebugNew replaces the standard `new` and `delete` operators as well as the array allocation/deallocation operators `new[]` and `delete[]`.

DebugNew provides the following services:

- Zaps newly allocated and free blocks with bad values to detect uses of uninitialized or deallocated memory.
- Addresses of blocks to be freed are checked to see if they are within the application heap.
- Tags allocated blocks to detect double freeing, freeing unknown blocks, and overwriting past the end of a block.
- A counter is incremented on allocations and decremented on frees. A warning is generated if there are more frees than allocs. You can inspect the counter as a crude check to see if any memory has leaked.
- Tracks the current number of bytes allocated via operator new and the maximum number of bytes ever allocated. Useful for tuning memory management and determining a good size memory pool to pre-allocate at program startup.
- Optionally tracks memory allocations and can report memory leaks. The source file and line is tracked for each allocation, and the list of allocated blocks can be dumped to a text file. Full use of this feature requires using the macro `NEW` instead of `new` in your source code.
- Optionally zaps and prevents reuse of freed blocks. This can help catch uses of dangling pointers.
- Tracks mismatching of regular and array `operator new` and `operator delete`, such as calling `delete[]` on a block allocated with `new`.

This section discusses the following topics:

- Setting Up DebugNew
- Using DebugNew
- Leaks.log

Setting Up DebugNew

The DebugNew source files are installed automatically by the CodeWarrior Installer. DebugNew is installed in `Metrowerks CodeWarrior:MacOS Support:Libraries:DebugNew`.

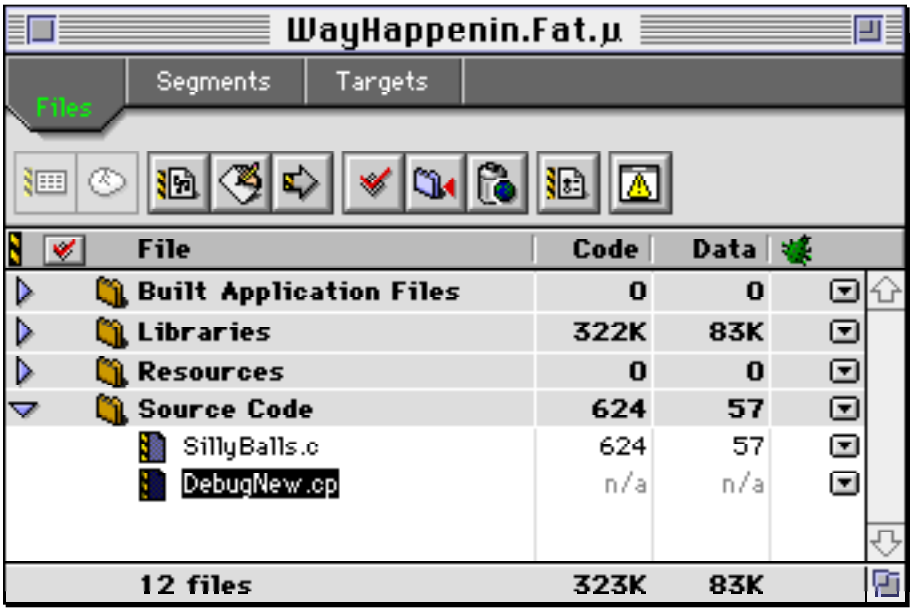
DebugNew is easy to setup. First, add `DebugNew.cp` to your current C++ or PowerPlant project, as shown in Figure 21.1. Next, you include the following line of code in every source file in your project:

```
#include <DebugNew.h>
```

The best method is to use a prefix file or precompiled header file. The method you use depends on the size of the project. If your project is small, use a prefix file in the C/C++ Language preference panel. If your project is large, like a PowerPlant project, use your own precompiled header file in the C/C++ Language preference panel.

To learn more about how to use precompiled header files, refer to the *IDE User Guide* documentation on the CodeWarrior CD.

Figure 21.1 Adding DebugNew



Using DebugNew

This section discusses, in more detail, how to setup your prefix or precompiled header file to use DebugNew for your project. DebugNew provides the following two modes of operation: basic and leak detection. The leak detection mode is described in this section.

This section discusses:

- Limitations
- Error Reporting
- Leak Detection

Limitations

By default, DebugNew only works for global operator new. However, if your class-specific operator new calls the global operator new, then those allocations will be tracked.

DebugNew does not attempt to do anything when allocation fails. That is what `DebugNewSetErrorHandler()` is for. Refer to "Advanced Techniques" on page 387 for more information.

Error Reporting

Errors are reported in two ways. When a validation check occurs at runtime, an error handler routine is called. The default routine just issues a `DebugStr()` call with a diagnostic message. If you prefer different behavior, you can provide your own error handler by calling `DebugNewSetErrorHandler()`. Validation errors occur when operator `delete` is called, or when your code calls `DebugNewValidatePtr()` or `DebugNewValidateAllBlocks()`.

Because DebugNew uses `DebugStr()` traps, you should have a low level debugger such as MacsBug installed on your system, or run your program in the IDE debugger with appropriate preferences set. For more information on this topic, refer to the *Debugger User Guide* for more information on `DebugStr()` traps.

Leak Detection

To enable full leak checking, you must `#include <DebugNew.h>` in every source file in your project, set up DebugNew's leak detection mode, and finally, replace every non-array usage of global operator `new` to use the macro `NEW`. This is best achieved by using a prefix or precompiled header file with some preprocessor directives.

Place the source code in Listing 21.1 at the top of your prefix or precompiled header file.

Listing 21.1 Enabling DebugNew

```
#define DEBUG_NEW 2   //enable leak detection
#include <DebugNew.h>

#define Debug_Version 1  // App is a debug version

#if Debug_Version
```

```
   #define new NEW  // use DebugNew's NEW macro
#endif
```

NOTE: The `Debug_Version` preprocessor variable, used throughout the examples, is not necessary to use DebugNew. However, it is good programming practice to conditionalize your debugging code.

WARNING! Use `#define new NEW` at your own risk because the compiler's preprocessor will indiscriminately replace every instance of `new` with `NEW`. This could cause undesirable side effects to occur. If you wish to use DebugNew's leak checking capabilities, it is recommended that you manually replace each instance of `new` you wish to track in your code with `NEW`. See the DebugNew Demo on the Reference CD for more information.

You can disable DebugNew by changing two lines as follows:

```
#define DEBUG_NEW  0   //disable DebugNew
#define Debug_Version 0  //App is not debug version
```

Using this method, you can continue to use DebugNew throughout the development cycle, turning it on or off as required, and then disable it (or remove it completely) for your final build.

Memory leaks are reported via the file "leaks.log". This file is written to the application's directory whenever `DebugNewReportLeaks()` is called. This call should be the last statement in `main()`, before your application exits, as shown in Listing 21.2.

Listing 21.2 Creating a log file

```
//=========================================================
//   • main
//=========================================================
//Do application initialization, create the application object,
```

```
//and run the main event loop.

void main()
{

  // Initialize memory and toolbox.
  InitializeHeap(5);
  UQDGlobals::InitializeToolbox(&qd);

  // Set up grow zone.
  LGrowZone grow(20000);

  // Bracket here so DebugNew can catch application-level leaks.
  {
    CTestApp  theApp;
    theApp.Run();
  }

  // If debugging, it's time to report on leaked C++ objects.

  #if Debug_Version

    DebugNewReportLeaks(); // dump leak info to leaks.log file

  #endif

}
```

Compile and run your program, performing various operations that you would do in a normal test cycle. When you quit your program, the leaks.log file will be created for you to examine.

Leaks.log

The leaks.log file provides useful information on where leaks are occurring in your code. Listing 21.3 shows a normal leaks.log file.

Listing 21.3 Normal leaks.log file

```
Maximum #bytes allocated at any point via operator new: 141284
There are 6 memory leaks, totaling 136 bytes:
  LComparator.cp line: 76, size: 4
  LDragAndDrop.cp line: 418, size: 40
  LPeriodical.cp line: 124, size: 40
  LComparator.cp line: 116, size: 4
  <unknown>, size: 40  // array
  LApplication.cp line: 134, size: 8
```

The leaks shown in Listing 21.3 are memory allocations that are only deleted upon quitting the application. These leaks are *normal*. If, however, leaks.log is a large file showing many leaks, then there is a serious problem you need to fix.

The file name and line numbers are provided by DebugNew's leak detection mode. Leak checking imposes a 24 byte overhead to each allocated block. There is also the overhead of finding and removing each block from the block list when it is deleted. The memory and time hit will be noticeable on moderate to large programs. Leak testing should generally only be enabled periodically to clean up leaks, and then turned off again.

TIP: Use MW ZoneRanger along with DebugNew to get more information on how your program is allocating and deallocating memory. To learn more about ZoneRanger, refer to the *CodeWarrior ZoneRanger Manual* on the CodeWarrior CD.

Advanced Techniques

One of DebugNew's more advanced features is the ability to install your own error handler. You can then handle various errors however you want. To install your own error handler, call `DebugNewSetErrorHandler()` as shown in Listing 21.4.

Listing 21.4 Installing an error handler

```
// Replace DebugNew's error handler.

  #if Debug_Version
    DebugNewSetErrorHandler(BetterDebugNewErrorHandler);
  #endif
```

Then you actually write your own error handler. The best method of doing this is to copy DebugNew's error handler into your own source file to use it as a shell. You then write code to expand on each condition you wish to handle. Listing 21.5 shows an example error handler you could write using PowerPlant's `Debug_Signal` mechanism.

Listing 21.5 A different error handler for DebugNew

```
  // ---------------------------------------------------------
  //  • BetterDebugNewErrorHandler
  // ---------------------------------------------------------
  //A replacement for DebugNew's error handler. Makes it easier to
  //identify the cause of the error in the debugger.

  void BetterDebugNewErrorHandler(short err)
  {
    switch (err) {
      case dbgnewNullPtr:
        SignalCStr_("DebugNew: null pointer");
        break;

      case dbgnewTooManyFrees:
        SignalCStr_("DebugNew: more deletes than news");
        break;

      case dbgnewPointerOutsideHeap:
        SignalCStr_("DebugNew: delete or validate called for
pointer outside application heap");
        break;

      case dbgnewFreeBlock:
```

```
        SignalCStr_("DebugNew: delete or validate called for free
block");
        break;

      case dbgnewBadHeader:
        SignalCStr_("DebugNew: unknown block, or block header was
overwritten");
        break;

      case dbgnewBadTrailer:
        SignalCStr_("DebugNew: block trailer was overwritten");
        break;

      case dbgnewBlockNotInList:
        SignalCStr_("DebugNew: block valid but not in block list
(internal error)");
        break;

      case dbgnewFreeBlockOverwritten:
        SignalCStr_("DebugNew: free block overwritten, could be
dangling pointer");
        break;

      default:
        SignalCStr_("DebugNew: undefined error");
        break;
    }
  }
```

22

Mac OS Support Files

This reference chapter tells you how to use libraries, shared libraries, header files, and interface files (including Apple's Universal Headers and Interfaces) with CodeWarrior Mac OS projects.

Topics discussed are:

- Using Libraries
- Using Shared Libraries
- Using Apple's Universal Interfaces
- Runtime Libraries
- Specifying Entrypoints
- Multiprocessor Development

Using Libraries

Libraries are special files that contain a set of routines used by your program. For example, the `MacOS.lib` library (for 68K projects) and the `InterfaceLib` shared library (for PowerPC and CFM68K projects) contains a set of routines that let you call into the Mac OS Toolbox.

When you create either a 68K or PowerPC project, CodeWarrior copies a library's object code into your project during linking. Any code that is not needed by the linked code will be stripped by the linker, so your project code will be as small as possible.

To learn about how to create your own static libraries, refer to "Creating Static Library Projects for Mac OS" on page 75.

To learn how to add library files to projects, refer to the *IDE User Guide*.

Using Shared Libraries

Like libraries, shared libraries contain a set of routines used by your program. The main difference from static libraries is that the code from shared libraries is not copied into your project when it is compiled, it is instead loaded dynamically at run time.

For example, when you use a shared library to create an application, you must place the shared library in the System Folder or in the same folder as the finished application at install time for your software. Then, when CodeWarrior launches the application and calls a routine from the shared library, the routine is executed using the object code in the shared library.

This method of accessing routines means that many different applications can access the same routines from the same shared library.

NOTE: Shared libraries are also known as Import Libraries and Dynamically Linked Libraries.

To learn about how to create your own shared libraries, refer to "Creating Shared Library Projects for Mac OS" on page 87.

Shipping Binaries Built With Shared Libraries

Unlike binaries built with libraries, such as applications and code resources, binaries built with shared libraries require some special handling. When you ship a binary created with a shared library, you must either:

- Ship the shared library with your binary, or
- Make sure the shared library you are using is one that is already in the user's System Folder or application folder, such as the AppleScript shared library from Apple computer.

CFM68K shared libraries built for 68K computers will only run on a 68K Mac OS computer (not a PowerPC Mac OS computer) that already has the `CFM-68K Runtime Enabler` extension installed. Your install procedure for your software must put this file in the System Folder of the 68K computer in order to use CFM68K shared libraries on 68K machines.

NOTE: Apple added the CFM68k Runtime Enabler to Mac OS 8.0.

Import "weak"

When a PowerPC or CFM-68K application loads, the operating system attempts to resolve imported symbols from all shared libraries included in the application's project.

For example, when an application calls the `NewHandle()` routine, the operating system searches for (on PowerPC machines) the `InterfaceLib` shared library.

- If `InterfaceLib` is found, the operating system links this routine to your application.
- If `InterfaceLib` is not found, the reference to `NewHandle()` is unresolved and the application is not launched.

When Import "weak" is used, the operating system launches the application whether or not one of its shared libraries exists. This leaves references your application makes to routines in the missing shared library unresolved.

To learn how to set the Import "weak" option, refer to the *IDE User Guide*.

NOTE: If you are using the import weak option and you attempt to call a routine from a missing shared library, your application will crash. While using import weak, you must verify that the routine is available by using the `Gestalt` function (most system services) or by comparing the import's address to `UnresolvedSymbolAd-`

`dress`. For more information, consult *Inside Macintosh: PowerPC System Software*, published by Addison-Wesley.

Initialize Before

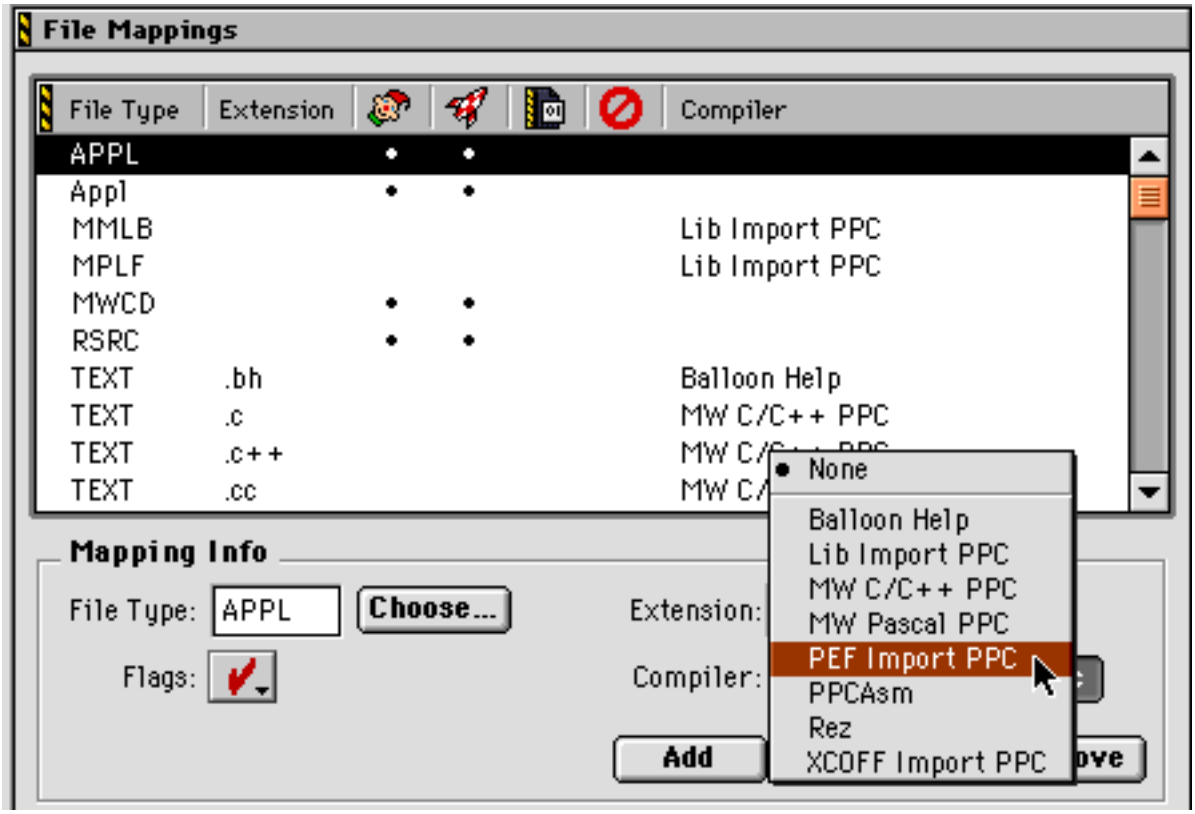
Initialize Before indicates which Program Executable Format (PEF) container (shared library or application) gets initialized first. By default, PEF containers being imported are initialized before the PEF containers importing them. This option may be used for mutually-dependent PEF containers to specify which gets initialized first.

Applications As Shared Libraries

The CodeWarrior compilers let you add applications (files with the APPL file type) to your project. If this file has a no data fork, it will be treated as a resource file and the resource fork will be copied into the PowerPC application during linking. That is one way to create "fat binaries".

If the application contains a data fork, however, you should link in the application as a shared library. To do this, use Set Current Target to select the target you wish to change this setting for, then go to the **File Mappings options** panel in the Target Settings dialog box. As shown in Figure 22.1, change the Compiler setting from None, to PEF Import PPC, and click the Change button. This setting will not affect other CodeWarrior targets, only the active project's targets are modified.

Figure 22.1 Import PEF PPC Setting



Using Apple's Universal Interfaces

The default headers included with CodeWarrior are Apple's most recent Universal Headers and Universal Interfaces.

The most important changes made to these headers in the past few years are:

- All QuickDraw globals (including `thePort`, `randSeed`, `ScreenBits`, `arrow`, `dkGray`, `ltGray`, `gray`, `black`, and `white`) are part of the struct qd and you must use qd. when referring to them. For example, use `qd.thePort` instead of `thePort`.
- `GestaltEqu.h` has been changed to `Gestalt.h`.

- If you are updating your projects from an older version of the CodeWarrior product, you **must** rebuild any Precompiled headers to avoid error messages. CodeWarrior' MacHeaders and PowerPlant precompiled headers on the CodeWarrior CD disc are already updated, but all others must be re-precompiled. Refer to the *IDE User Guid*e for information on using .pch files for updating precompiled headers automatically.
- A "Length" macro is now used in MacHeaders, so you will get odd error messages if you have a variable with that name.
- You may get new errors if you have been declaring C/C++ variables with `const` and `volatile` freely.

C/C++ Headers

Header files are specified by using the `#include` compiler directive. For example, if you are compiling a C program that uses the Sound Manager routines, add the `MacOS.lib` to your 68K project (`InterfaceLib` to your PowerPC project) to get access to the Mac OS Toolbox routines. Then include the following line to specify that your source code file uses Sound Manager routines:

```
#include <Sound.h>
```

The `Sound.h` header file defines all of the sound routines available from the Sound Manager.

Most Mac OS Toolbox routines are already included in your project. To learn more about Prefix Files, refer to "Prefix File" on page 397.

Pascal Interfaces

Interface files are specified by using the USES clause. For example, if you are compiling a Pascal program that uses the Sound Manager routines, add the file named `MacOS.Lib` to your 68K project (use the file named `InterfaceLib` for your PowerPC projects) to get access to the Mac OS Toolbox routines. Then include the following line to specify that your source code file uses Sound Manager routines:

```
USES Sound;
```

The `Sound.p` interface file defines all of the sound routines available from the Sound Manager.

Prefix File

In the C++ and Pascal Language options panels, you can use the Prefix File text entry field, for both the C/C++ and Pascal compilers, to specify a file that is read and compiled before any file in the open project. By default, CodeWarrior C/C++ compilers set this entry field to either `MacHeaders68K` (for 68K) or `MacHeadersPPC` (for PowerPC).

These prefix files are precompiled header files that already include declarations for many of the Mac OS Toolbox headers. By precompiling all the common declarations into a single Prefix File, the compiler can more quickly compile your source code files.

For information on creating your own precompiled header files or for instructions on recompiling `MacHeaders`, consult the section on Precompiled Headers in the IDE *User Guide*.

Runtime Libraries

This section describes the CodeWarrior PPC (PowerPC) runtime libraries. The information presented here is similar for CFM68K libraries (except the library names).

After reading this section, you should refer to "Specifying Entry-points" on page 399 for more information you will need to correctly configure your projects.

Topics discussed include in this section include:

- About Runtime Libraries
- Exception Handling
- Choosing a Runtime Library

About Runtime Libraries

The runtime library is a collection of routines and data that implement various aspects of the runtime environment, including:

- Exception handling
- Runtime type information (RTTI)
- Operator new and operator delete
- Setjmp() and longjmp()
- Initialization and destruction of static objects
- Pointers to member functions
- Program startup and termination sequences

Virtually all programs use some of the routines in the runtime library.

Exception Handling

The exception handling implementation must keep track of all loaded code fragments that make up a program. The runtime library provides the data structures and operations for doing this.

This information is usually registered and unregistered as a code fragment is loaded/unloaded. If it is not properly registered, the exception handler will fail to find any handlers for a thrown exception.

To ensure that exceptions work correctly in the presence of shared libraries, you must follow the instructions provided in the section entitled "Specifying Entrypoints" on page 399.

Choosing a Runtime Library

To learn how to choose libraries to link with application projects, refer to "Choosing Libraries for Applications" on page 62. To learn how to choose files to link with shared library projects, refer to "Choosing Libraries for a Shared Library Project" on page 96. To learn how to choose files to link with code resource projects, refer to "Choosing Libraries for Code Resource Projects" on page 121. To

learn how to choose libraries to link with MPW Tools that you are creating, refer to "Choosing Libraries for an MPW Tool Project" on page 147.

Specifying Entrypoints

In addition to choosing the proper libraries, it is important to specify the correct entrypoint(s) in the options panels.

The following sections provide entrypoints information for the following listed program types:

- Building a Monolithic Application
- Building a Monolithic Shared Library or Drop-in
- Building a Multi-Fragment Application
- Building Multi-fragment Shared Library
- Building a Multi-Fragment Drop-in
- Installing Initialization and Termination Entrypoints
- Using ANSI C/C++ Libraries

Building a Monolithic Application

To build a monolithic application, make sure that you include the correct libraries, as discussed in the section entitled "Choosing Libraries for Applications" on page 62.

Then, specify the entrypoints shown in Table 22.1 in the PPC Linker or CFM68K Target Settings panel.

Table 22.1 Monolithic application entrypoints

Initialization:	`<empty>`
Main:	`__start`
Termination:	`<empty>`

The `__start()` routine handles all the required initialization, including:

- Registering the code fragment with the exception-handling mechanism,
- Calling all static initializers, setting up 'argc' and '`argv`', calling `main()`, and
- Calling `exit()` to terminate the program, destroy static objects, etc.

Building a Monolithic Shared Library or Drop-in

To build a monolithic shared library or drop-in, make sure that you include the correct libraries, as discussed in the section entitled "Choosing Libraries for a Shared Library Project" on page 96.

Then, specify the entrypoints shown in Table 22.2 in the PPC Linker or CFM68K Target Settings panels.

Table 22.2 Monolithic shared library or drop-in entrypoints

Initialization:	`__initialize`
Main:	`<empty>`
Termination:	`__terminate`

In this case:

- The `__initialize()` routine registers the code fragment with the exception-handling mechanism and calls static initializers.
- The `__terminate()` routine destroys static objects and unregisters the code fragment.

NOTE: This kind of code fragment must not throw exceptions outside of the shared library, and should not call `exit()` or `abort()`.

If your shared library depends on other shared libraries (besides the usual Mac OS Toolbox interfaces), it is to your advantage to make it a multi-fragment shared library, as described in the "Building a Multi-Fragment Application" on page 401.

Building a Multi-Fragment Application

A multi-fragment application consists of the main application, one or more shared libraries, and possibly some drop-ins that are loaded and unloaded while the program is running. No shared libraries or dopins are necessary to make a multi-fragment application. It is the runtime in a shared library that makes an application multi-fragment.

In this case, it is important to have a single implementation of many of the runtime support routines, shared by all the fragments that make up the program.

Also, if a multi-fragment application uses the ANSI C or C++ libraries, it should use them as shared libraries rather than linking a distinct copy of the libraries into each fragment.

To build a multi-fragment application, make sure that you include the correct libraries, as discussed in the section entitled "Choosing Libraries for Applications" on page 62.

Then, specify the entrypoints shown in Table 22.3 in the PPC Linker or CFM68K Target Settings panels.

Table 22.3 Multi-fragment application entrypoints

Initialization:	`__initialize`
Main:	`__start`
Termination:	`__terminate`

In this case:

- `__initialize()` registers the code fragment with the exception-handling mechanism and calls static initializers

- `__start()` sets up '`argc`' and '`argv`', calls `main()`, and calls `exit()` to terminate the program, destroying static objects in all loaded fragments
- `__terminate()` unregisters the code fragment.

Unlike with a standalone application, you must specify the initialization and termination entry-points, or exception-handling will fail.

Building Multi-fragment Shared Library

To build a multi-fragment shared library, make sure that you include the correct libraries, as discussed in the section entitled "Choosing Libraries for a Shared Library Project" on page 96.

Then, specify the entrypoints shown in Table 22.4 in the PPC Linker or CFM68K Target Settings panels.

Table 22.4 Multi-fragment shared library entrypoints

Initialization:	`__initialize`
Main:	`<empty>`
Termination:	`__terminate`

In this case:

- `__initialize()` registers the code fragment with the exception-handling mechanism and calls static initializers
- `__terminate()` unregisters the code fragment.

Global objects

Global objects that must be destroyed at program termination are registered in an application-wide list maintained by the runtime library. These objects are destroyed, in the proper order, when the main application calls `exit()` or returns from `main()`.

Take care when creating the exports file for your shared library. If you choose **All Global Symbols**, the result will be linker warnings

about duplicate definitions of some symbols, possibly including '`longjmp`', '`__ptmf_null`', '`__terminate`', and '`_initialize`'.

The best approach to creating the export file is to follow these steps:

1. **Select Use.exp File from the Export Symbols pop-up menu in the CFM68K or PPC PEF Target Settings panels.**
2. **Edit the generated.exp file. Use the pound sign (`#`) to comment out the names.**
3. **Add the generated.exp file directly to the project.**

WARNING! Don't forget this step, because the linker does not check the directory containing the project file.

4. **Re-link your project target.**

Building a Multi-Fragment Drop-in

To build a multi-fragment drop-in, make sure that you include the correct libraries, as discussed in the section entitled "Choosing Libraries for a Shared Library Project" on page 96.

Then, specify the entrypoints shown in Table 22.5 in the PPC Linker or CFM68K Target Settings panels.

Table 22.5 Multi-fragment drop-in entrypoints

Initialization:	`__initialize`
Main:	`<empty>`
Termination:	`__terminate`

In this case:

- `__initialize()` registers the code fragment with the exception-handling mechanism and calls static initializers.
- `__terminate()` destroys static objects and unregisters the code fragment.

Because a drop-in may be loaded and unloaded while the program is running, CodeWarrior cannot perform static objects destruction at program exit. Static object destruction must occur before the fragment is unloaded.

Installing Initialization and Termination Entrypoints

Sometimes you need to install your own initialization and termination routines, for example, to make use of the information that the Mac OS Code Fragment Manager provides.

If you are using something other than `__initialize()` and `__terminate()`, be certain that:

- The first thing your initialization entrypoint does is call `__initialize()`.
- The last thing your termination entrypoint does is call `__terminate()`.

Using ANSI C/C++ Libraries

If you are using the MSL libraries, make sure they appear in the link order **before** any of the runtime libraries in your project, such as `MWCRuntimePPC.lib` in the project window (or, for the MPW-hosted tools, in the linker command line). These libraries provide implementations of the `exit()` and `abort()` functions that must override the versions in the runtime library. Otherwise, the MSL libraries will not properly flush buffers, close open files, or close the Console window.

To move library files around in the link order, you just drag and drop them in the Link Order view of the project window, as shown in Figure 22.2 and 22.3. Moving files to the top of the list moves them further toward the front of the link order. Moving files lower in the list moves then toward the end of the link order.

Figure 22.2 Link Order View of Project Window

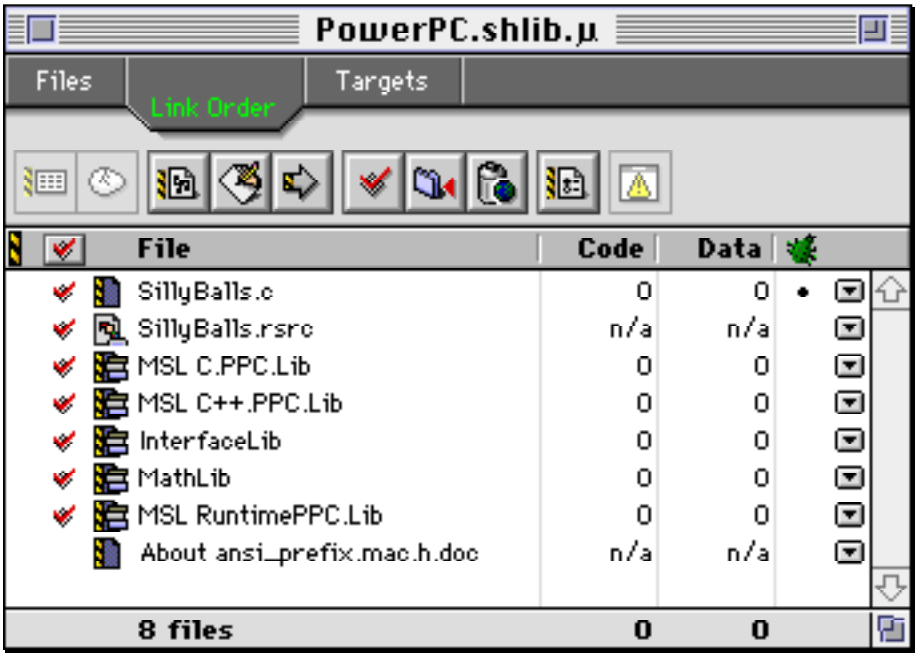
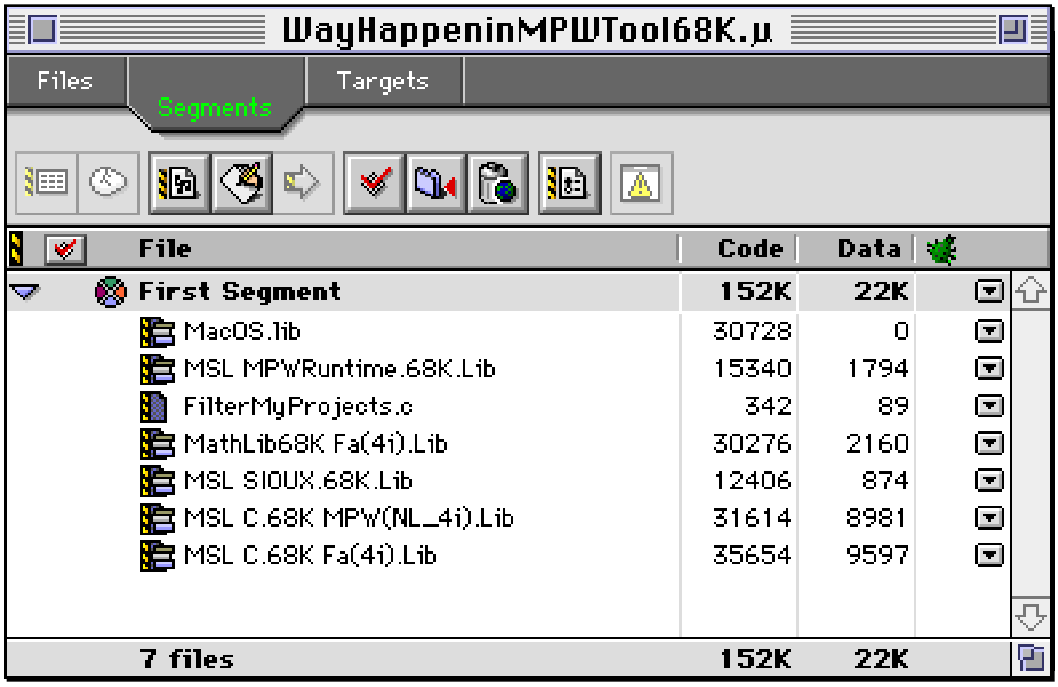


Figure 22.3 Link Order view of Project Window



Multiprocessor Development

To learn more about how to develop code for Mac OS computers that contain more than one PowerPC processor, refer to the resources included in the `Metrowerks CodeWarrior` folder. In the `Mac OS Support:Headers:System Extras Headers` folder you will find a folder named `Multiprocessing`. This folder contains a header file you will need to include, a library file that must be installed on your Mac OS multiprocessing computer, and other documentation to help get you started.

You will also find more information on multiprocessor development on the Apple Developer CD's, or on the internet world-wide web at `http://www.apple.com`/developer

23

Troubleshooting

This chapter discusses various problems people have encountered while writing and debugging their code. There are suggested solutions for each problem. If your problem is not in this chapter, please contact Metrowerks Technical Support for assistance. This chapter will expand to include more problems with each release of CodeWarrior.

The topic areas covered include:

- Programming Specific
- IDE Specific
- Debugger Specific
- Debugger Error Messages
- Working With Other Debuggers

Programming Specific

This section discusses some problems that are commonly encountered when writing code for the Mac OS. This section continues to expand as more problems are encountered.

This section has one primary topic:

- BowelsOfTheMemoryMgr

BowelsOfTheMemoryMgr

This problem and answer, by Apple Developer Support, appeared in the "Q &A" section of the August 1997 issue of *MacTech* magazine (Vol. 13. No. 8). It is reprinted here with permission. Metrowerks thanks the staff at *MacTech* for use of this question.

Problem

Sometimes, MacsBug generates a PowerPC unmapped memory exception with an address that starts with the symbol `BowelsOfTheMemoryMgr`. What does this mean?

Solution

When you are displaying addresses in MacsBug, MacsBug shows offsets from the last symbol it can find. In the Modern Memory Manager on Power Macintosh computers up to Mac OS version 7.6, the last symbol was `__HSetStateQ`. The code after `__HSetStateQ` consists of various internal Memory Manager subroutines. So, if there's a hang or crash in an internal Memory Manager subroutine, it shows up ain MacsBug as `__HSetStateQ+xxxxxxxx`.

Various system software engineers were tired of seeing bug reports that said __HSetStateQ was crashing, so beginning with Mac OS 7.6, we decided to add a new last symbol to the Memory Manager. As a consequence, bug reports would be somewhat more informative. We thought of naming the new symbol `YourHeapIsCorrupt` (since that's usually the case when a program crashes the Memory Manager) but decided on `BowelsOfTheMemoryMgr` instead because that's where you are.

So if you're crashing or hanging at `BowelsOfTheMemoryMgr+xxxxxxx`, type **`HC`** to see if your heap is corrupted (it probably will be) and then start debugging your code to find out how it got corrupted.

IDE Specific

This section discusses problems encountered while using the CodeWarrior IDE on Mac OS computers.

At the time of this writing, there are no problems to report. However, this section may expand in the future.

Debugger Specific

This section discusses problem you may encounter while using the IDE debugger on the Mac OS. Problem topics include:

- General Problems
- The debugger won't launch
- Debug does nothing
- Debugger doesn't Regain Control after my Exception Handler is Called
- Falling into MacsBug
- Crashing in the Drag Manager
- The debugger crashes
- Debugger() and DebugStr()
- SysBreak() and SysBreakStr()
- Pointer variables have a "Bus Error" value

General Problems

There may come a time when one of the solutions in this section doesn't seem to work, or your specific problem isn't in here. before sending a note to Metrowerks Technical Support, try:

- Remove any remaining Debugger Inits, remove DebugServices (Startup Items folder), make sure you have only one copy of MetroNub (latest version), remove RamDoubler (older than 1.5.2), remove "[space]ProcessManagerInit" on 68K Macintoshes

The debugger won't launch

Problem

Even if **Enable Debugger** is selected, when I run my application, the debugger doesn't launch.

The **Run** or **Debug** command in the CodeWarrior Project menu is dimmed.

Background

If your project compiles code for the PowerPC, you cannot run that code on a 68K Mac.

Solutions

- Make sure you aren't trying to launch a PowerPC program on a 68K Macintosh. Although Power Macintosh computers can run both 68K and PowerPC programs, 68K Macintosh computers can run only 68K programs.
- You may be missing `ObjectSupportLib`. You can install it from your CodeWarrior CD. It is installed automatically by System 7.5 or later.

Debug does nothing

Problem

The **Debug** command does nothing.

Background

you must have everything set up properly for the debugger to work correctly. In addition, make sure your code actually does something!

Solutions

- On 68K Mac OS computers, set 32-bit addressing on in the Memory control panel and restart your computer. The debugger cannot debug your program in 24-bit addressing mode.
- Turn off virtual memory and restart your computer. The debugger nubs may be incompatible with virtual-memory management system extensions on both 68K and Power Macintosh computers.

Debugger doesn't Regain Control after my Exception Handler is Called

Problem

My code currently calls `InstallExceptionHandler()` with my exception handler. The handler currently assumes any exception is fatal, and aborts (after some cleanup and an appropriate message) if the exception type is not `trapException` or `traceException`; otherwise, it just returns `noErr`.

When I step through my code and try to step over the `InstallExceptionHandler()` call, the debugger never regains control, and my application hangs.

Solution

The problem is you are returning `noErr` from your exception handler. You should return `noErr` only if you completely handled the exception and want to resume execution. If you do not handle the exception, return a non-zero value and the next exception handler in the chain (MetroNub or MacsBug) will get a chance to handle the exception.

Listing 23.1 shows what your exception handler should look like.

Listing 23.1 How to Handle Exceptions

```
OSStatus MyExceptionHandler(ExceptionInformation *theException)
{
  if (theException->kind == trapException ||
    theException->kind == traceException)
      return -1;//let the debugger handle the exception
  else
  {
    ... // whatever your handler wants to do
  }
}
```

Falling into MacsBug

Problem

I'm working on a Power Mac, and I'm always breaking into MacsBug when the debugger is running. MacsBug 6.5d12 continually comes up with a bus error or an illegal instruction when nothing appears to be wrong.

Background

This is not a bug in the debugger—it is just doing what debuggers need to do. Unfortunately, MacsBug 6.5d12 doesn't take kindly to it.

This version of MacsBug intercepts CPU exceptions caused when the debugger reads from invalid memory addresses. The debugger reads from such addresses as a regular part of its job. The debugger installs its own CPU exception handler so it can recover gracefully if an address is bad. However, MacsBug 6.5d12 is getting control and reporting the message before the debugger's handler gets a chance to run. MacsBug will break with a message like "PowerPC unmapped memory exception" or "PowerPC access violation exception."

Solutions

- Use a newer version of MacsBug. As of this writing, the current version of MacsBug is 6.5.4, and is available free from Apple's web servers and FTP sites.

Crashing in the Drag Manager

Problem

Why do I crash when I try to debug inside a Drag Manager callback routine?

Background

Debugging a Drag Manager callback is hard because the Drag Manager disables process switches. As a result, the debugger nub can't get control reliably. The latest nubs and system help this somewhat.

Solutions

- Use a low-level debugger.
- Debug "at a distance." Inside the callback routine under development, use `SysBeep()` calls and other tricks to tell yourself how far along you are and what's going on.

The debugger crashes

Problem

The debugger issues an assertion failure, quits unexpectedly, or stops the computer completely.

Background

The debugger has control over your computer only while execution is taking place in the target program. The debugger has limited control over bugs or conflicts occurring in the Mac OS, system extensions, and interrupt handlers.

Solutions

- Make sure you are not stepping through any nonreentrant code, such as operating-system routines, drag handlers, or interrupt handlers.
- If you are debugging PowerPC object code, make sure your program properly uses universal procedure pointers (UPPs) when it calls or is being called by the Mac OS in a callback routine. Some OS routines require a Mode Manager switch. The Mixed Mode Manager requires a properly initialized UPP to successfully make a switch from PowerPC to 68K emulation and back.
- If you are debugging C++ object code and the debugger crashes before it reaches the `main()` function, check static constructor functions. Static constructors, called before the `main()` function, may be causing the crashes.
- The `.dbg` file associated with your project may be corrupted. The `.dbg` file holds a variety of information used by the debugger, like breakpoint locations and conditions, win-

dow positions, and so forth. Try throwing away and rebuilding your `.dbg` file.

- Debug the constructors for global objects. Global object constructors are executed before the debugger settles down with a screen showing the arrow at `main()`. If these have a bug, it can crash the debugger as your app starts up. You can insert calls to `Debugger()/DebugStr()` or `SysBreak()/SysBreakStr()` into your constructors to drop you into a debugger. You must have MacsBug installed for these calls to work.
- Increase the application memory partition of the application being debugged.

It is definitely possible that you can crash your program in a way that takes out the debugger too. For example, if you pass a bad pointer to `GetNewWindow()`, you could end up corrupting some of the Window Manager's internal data structures. You might end up trashing some low memory global variables that are read by the Mac OS at interrupt time. You could overwrite the 68K exception vectors, which would eliminate the possibility of any debugger handling the crash. Remember that your program, the debugger, and the OS all live together in the same address space with no protection of any kind in-between.

See also the *Targeting Mac OS* manual for more information on debugging code resources and static constructors.

`Debugger()` and `DebugStr()`

Problem

The `Debugger()` and `DebugStr()` calls are crashing my machine instead of dropping me into the debugger. Or I end up in a low-level debugger when I use these calls in 68K code.

Background

For these calls to work, *you must have a low-level debugger installed*. And even then, from 68K code they drop you into the low-level debugger instead of the CodeWarrior debugger.

Solution

- To intercept `Debugger()` and `DebugStr()` calls in 68K code, install a low-level debugger like MacsBug. The CodeWarrior debugger only intercepts `SysBreak()` and `SysBreakStr()` calls in 68K code.

See also "Stopping Under Program Control" on page 359.

`SysBreak()` and `SysBreakStr()`

Problem

The `SysBreak()` and `SysBreakStr()` calls are crashing my machine instead of dropping me into the debugger. Or I end up in a low-level debugger when I use these calls in PowerPC code.

Background

For these calls to work, *you must have a low-level debugger installed*. And even then, from PowerPC code they drop you into the low-level debugger instead of the CodeWarrior debugger.

Solution

- To intercept `SysBreak()` and `SysBreakStr()` calls in PowerPC code, install a low-level debugger like MacsBug. The CodeWarrior debugger only intercepts `Debugger()` and `DebugStr()` calls in PowerPC code.

See also "Stopping Under Program Control" on page 359.

Pointer variables have a "Bus Error" value

Problem

When I stop in a routine, the debugger shows pointer variables with a value of "• Bus Error •."

Background

There are two reasons why a pointer variable may have a "• Bus Error •" value. The variable may not have been initialized

(so it has a random value, which is likely to be an invalid address). Or you may be using 24-bit memory addressing on a 68K Mac.

Solutions

- Continue stepping through the program to make sure the variable is assigned a value. Once a pointer variable has a value, the debugger will display it correctly. If the pointer variable is never initialized, you have found a bug in your code!
- Make sure 32-bit addressing is on in the Memory control panel, then restart your computer before debugging again. The debugger can only display 32-bit addresses.

Debugger Error Messages

This section briefly discusses various error messages you may encounter while using IDE debugger on the Mac OS.

An error occurred while creating or sending an AppleEvent

A catch-all message for an error in sending an AppleEvent to another application.

Can't send the AppleEvent because the file must be a text file.

Only text files are valid for the "Open in Editor" command.

Can't send the AppleEvent because the file was not found.

Source file not found during “Open in Editor” command.

Can't send the AppleEvent because the target application is not running.

The target application must be running in order to receive an AppleEvent.

Could not launch “^0” because the library “^1” was not found.

CFM applications used shared libraries. If a necessary library was not found, the application cannot be launched.

Could not startup because MetroNub is not installed.

The debugger cannot run without a nub, which must be installed in your system Extensions folder and loaded at startup time.

Out of memory!

The IDE debugger ran out of memory while debugging. Try increasing the memory allocation.

The debugger cannot run on this Macintosh. It requires System 7.1 or later and a 68020 processor or higher.

The executable file is missing some of the code resources specified by the SYM file. Please check that the correct executable file was selected.

Either the SYM file is out of date, or you have matched up the wrong SYM file and executable file.

The installed version of MetroNub is too new. Please update MW Debug, or use an earlier MetroNub.

The version of MetroNub in your system folder doesn’t match this version of the debugger.

The installed version of MetroNub is too old. Please update to the current version.

The version of MetroNub in your system folder doesn’t match this version of the debugger.

Working With Other Debuggers

This section covers problems using MW Debug, the standalone debugger, in conjunction with other debuggers. Problem topics include:

- Jasik's The Debugger
- QC Deactivates
- MacsBug stack crawl
- QC crashes when I try to use it

Jasik's The Debugger

Problem

When I run Jasik's The Debugger and the CodeWarrior debugger simultaneously, my computer crashes.

Background

These two debuggers are incompatible for debugging PowerPC code if you use the standard install options. They require different and conflicting low-level services.

Solutions

- MW Debug can work simultaneously with Jasik's debugger to some extent. In order to get this to work, rename MetroNub to zMetroNub so it installs after Jasik's xDBGR_Startup file.
- Use one and remove the other. To run the CodeWarrior debugger, remove any system extensions or control panels for Jasik's The Debugger. If you are using Jasik's The Debugger, do not launch the CodeWarrior debugger.
- For low-level debugging features to complement the CodeWarrior debugger in PowerPC code, install MacsBug on your computer.

QC Deactivates

Problem

The QC system extension from Onyx Technology deactivates after the debugger shows a QC error report.

Background

The debugger deactivates QC when it receives an error from QC. To reactivate QC, use the hot-key combination shown in the QC control panel.

Solution

- To stop the debugger from deactivating QC (and from displaying QC errors), deselect the **QC-aware** preference in the debugger's **Preferences** dialog box.
- If you are using MW Debug to catch QC detected errors, you will want to make sure that **Catch XXX Debugger and DebugStr traps** (where XXX is 68K or PowerPC) is turned on in the debugger preferences. QC will cause a user break when it reports an error. You can then use MW Debug to look back up the stack to find out where things left your code.

See also "MetroNub Settings" on page 357.

QC crashes when I try to use it

Problem

I can't get QC and the debugger to work together. I have the debugger preference **QC-aware** checked. I have QC's **Debugger breaks** checked. But when I know QC should inform me of a problem, my Mac crashes.

Background

This problem is caused by a bug in QC version 1.1.3. When virtual memory is in use, the debugger does not catch QC-detected errors.

Solutions

- Get a more recent version of QC, if available.
- Turn off virtual memory.

MacsBug stack crawl

Problem

MacsBug won't display a stack crawl for a PowerPC program.

Background

You are using an older version of MacsBug that was originally designed for 68K code only.

Solutions

- Install the PowerPC `'dcmd'` extensions to MacsBug. With the proper extensions installed, use the `scp` command to display the routine-call chain from MacsBug.
- Get the latest version of MacsBug, which is PowerPC-aware.

24

Library Naming Conventions

This section of the manual discusses how the names for Metrowerks libraries are determined.

Library Naming Conventions

If you look at the names of the library files furnished with the CodeWarrior product, you will find that you may need some help determining what each library does. You need this information if you are trying to build a project from scratch, or if you are trying to determine which library you should use to convert one of your old projects to MSL, or the Metrowerks Standard Libraries.

The general naming convention for MSL C and C++ is based on the following format:

```
MSL [language].[processor] [Far|MPW](compile settings).[A4
addressing for 68K].Lib
```

For example, a library might be named something like:

```
MSL C++.68K Fa(4i_8d).Lib
```

This nomenclature scheme is intended to quickly identify binaries for all the various MSL targets. It is also intended to allow for library names to be unchanged across all CodeWarrior host platforms, so no reserved characters are used in library names.

In addition to the rules presented here, it is important to note the following additional hints:

- Shared-Library binaries have the 3 character extension "DLL"

- Normal Library binaries have the 3 character extension "Lib"

Intel has its own naming convention which is followed for x86 libraries for Microsoft compatibility. Refer to the *Targeting Win32* manual on the CodeWarrior CD for additional information on this topic.

For all other targets, specific names are of the form indicated below.

```
MSL [C|C++].[68K|PPC|Mips] [Fa|MPW](NL_xi_F_8d).[A4].Lib
Where:
     C - standard language library for ISO C
   C++ - standard language library for draft ISO C++
   68K - Motorola 680x0 processors (68000, 68020, 68030, 68040)
   PPC - A/I/M processors using the PowerPC ISA (e.g. 403, 601,
         603, 603e, 603ev, 604, 604ev with Mac OS/PowerOpen ABI)
  Mips - Mips ISA (R3000, R4000)
    Fa - far data turned on, default is off
   MPW - built to use MPW entry points and signal naming
         conventions
    NL - built with "convert NL to CR" on, default is off
    xi - built with "x" size integers (i.e. 2 byte, 4 byte),
         default varies w/ processor (68K = 2, PPC = 4)
     F - 68K only, built with math coprocessor support activated
         (68881)
    8d - built with 8 byte double types, default varies
    A4 - applicable to 68K for code resources only,
         uses A4 addressing for data
```

Index

Symbols

#if defined(powerc) 41, 56
#pragma export 94
'????' 272
'APPL' 272
'RSED' 266
'rseg' resource 34, 262
'SIZE' Flags option, PPC Target panel 176
.dump 160
.exp file 92
.iSYM 356
.r files 156
.rsrc 159
.SYM 356
.xSYM 356, 362
@ operator 325
__abs() 245
__*cntlzw()* 245
__eieio() 244
__fabs() 245
__fnabs() 245
__fres() 246
__fsel() 246
__fsqrte() 246
__isync() 244
__labs() 245
__*lhbrx()* 245
__*lwbrx()* 245
__profile 379
__rlwimi() 247
__rlwinm() 247
__rlwnm() 247
__setflm() 246
__*sthbrx()* 245
__*stwbrx()* 245
__sync() 244
~ (C++). See destructors (C++)
'cfrg' 362
'MDEF' 364
'rsrc' 160
'shlb' 96, 272
'SIZE' resource 60
'XCMD' resources 363

Numerics

128-bit floating-point numbers 221
32-bit clean applications 72, 177, 258
4-Byte Ints option, 68K Processor panel 284, 306
4D extensions 363
68020 Codegen option, 68K Processor panel 282, 315
68881 329
68881 Codegen option 308, 315
68881 Codegen option, 68K Processor panel 285, 315
68881 Codegen preferences panel opption 329
68K 328
  about LONGINT and arithmetic speed 327
  array types 222, 330
  Calling conventions
    overview 333
    Simple type parameters 333, 334
  data types overview 325
  DOUBLE 329
  enumerations 332
  EXTENDED 329
  floating-point data types 328
  function results for 335
  INLINE directive 324
  INTEGER 326
  integer data types 326
  LONGINT 326
  LONGREAL 329
  PACKED array of CHAR 331
  PACKED arrays 222, 330
  Pascal/STRING 331
  REAL 329
  records 332
  registers and MacOS 338
  Sets 332
  SINGLE 329
  UNSIGNEDBYTE 327
  UNSIGNEDLONG 328
68K A4-based application 76
68K A5-based application 76
68K Disassembler panel 288–290
68K Linker panel 291–294
68K Processor panel 278–288
68K Project preference panel

code resource 114
shared library 95
68K settings panels *See* settings panels
68K Target panel 255–278
8-Byte Doubles option 284, 285, 307

A

A4 globals 363
A4 register 76
A4 Relative Data 80, 84
A4 relative data option, 68K Target panel 270
A4Stuff.h 124
A5 globals 363
A5 register 76
A5 Relative Data 80, 84
A5 relative data option, 68K Target panel 270
`__abs()` 245
accelerated code resource 103
acceptAppDiedEvents 72, 177, 258
acceptSuspendResumeEvents 72, 177, 258
Add Files 49, 58
All Globals 95
allocating variables for 68K 310
AppDied events 72, 177, 258
Apple Events, handling 73, 177, 178, 259
Apple Universal Headers 395
applications
68K 34
background 72, 177, 258
CFM68K 34, 262
changing Creator 69
changing memory 70
changing name 69
changing Type 69
choosing libraries for 62–68
creating Fat 39–62
creating projects 35–67
Fat 34
PowerPC 34
PPC 34
setting options 68–74
types of 33
arguments
passing in registers 304
ARRAY 337
Array types 222, 223, 330, 331
Arrays 222, 330
`asm` blocks not supported 230, 340
`asm` keyword 230, 340
assembler
stand-alone described 27
See also inline assembler
Automatically Launch Applications When SYM File Opened 371

B

back-end compiler *See* compiler
background application 72, 177, 258
backup copy 42
`bfchg` assembly statement 349
`bfclr` assembly statement 349
`bfexts` assembly statement 349
`bfextu` assembly statement 349
`bfffo` assembly statement 349
`bfins` assembly statement 349
`bfset` assembly statement 349
`bftst` assembly statement 349
bitfields
size of 310
`bool` size 209, 306
BOOLEAN 333, 334, 336
Break on C++ Exception 374
breaking into the debugger 359
breakpoint
runtime 359, 360
building 39
bus error 415

C

C++
debugging 367–374
exceptions, debugging 374
static constructors 371, 413
static destructors 372
C++ headers 396
C/C++
headers 396
Calling conventions 225, 226, 333

calling conventions
registers 304
calling conventions for 68K 309–310
calling conventions for PowerPC 210–212
`CallUniversalProc()` 128
canBackground 72, 177, 258
catch block, debugging 374
CDEF 101
CFM68K 34, 40, 71, 77, 80, 84, 262, 263, 362, 393
applications 77
CFM68K library option, 68K Target panel 270
CFM68K panel 294–298
CFM-68K Runtime Enabler 34, 262, 393
CHAR 334
`char` size 209, 306
Chooser extensions 101
class specialization
and debugging 373
__cntlzw() 245
Code Fragment Manager 34, 262, 362
Code Model option, 68K Processor panel 279
code resource 122
68K globals 123
accelerated 103
calling PowerPC 128
choosing libraries for 121–122
creating 110–129
creating 68K 104–107
creating multi-segment 110–114
creating PowerPC 107–110
creating single-segment 110–114
custom headers 124
Fat 103
globals 123
libraries for 68K 121
safe Fat 103
setting options 114–119, 263–265
types of 101
writing 123–129
Code Sorting option, PPC PEF panel 201
`code68020` pragma 315
`code68881` pragma 316
CodeWarrior
compare to command line 28
compiler architecture 24
compiler described 26
components 25–28
debugger described 27
described 17
development process 28–31
documentation architecture 18
IDE described 25
installing 24
linker described 27
stand-alone assembler described 27
tools listed 25
Collapse Unused TOC-reloads option 203
command line 63
command-line and CodeWarrior compared 28
comments in inline assembler 234, 342
compiler
architecture 24
back-end for 68K 301–319, 323–338
back-end for PowerPC 207–215
described 26
other documentation 208, 301, 323
support for inline assembly 229, 339
See also C Compilers Reference
See also Pascal Compilers Reference
compiling 30
Composite parameters
and PowerPC calling conventions 226
configuring network 355
console.stubs.c 63
Constructor 155
constructors (C++) 371
debugging 369
Control Definition 101
Control Panel 101
Create New Target 42
Creator field 69, 108, 116
Creator option, 68K Target panel 260, 266, 272
Creator option, PPC Target panel 167, 178
cross debugger 353
C-style strings 223, 331
Current Version option, CFM68K panel 203, 297
Current Version option, PPC PEF panel 203
CustomCRStartup.lib 125

D

data fork 34
Data types
  array types
    68K 222, 330
  C/PACKED array of CHAR
    68K 331
    PowerPC 223
  DOUBLE for 68K 329
  DOUBLE for PowerPC 221
  enumerations 224, 332
  EXTENDED for 68K 329
  floating-point types for 68K 328
  INTEGER for 68K 326
  INTEGER for PowerPC 218
  integer types for 68K 326
  integer types for PowerPC 218
  LONGINT for 68K 326
  LONGINT for PowerPC 218
  LONGREAL for 68K 329
  LONGREAL for PowerPC 221
  overview for 68K 325
  PACKED arrays
    68K 222, 330
  Pascal/STRING
    68K 331
    PowerPC 222
  REAL for 68K 329
  REAL for PowerPC 221
  records 224, 332
  sets on 68K 332
  sets on PowrePC 223
  SINGLE for 68K 329
  SINGLE for PowerPC 221
  UNSIGNEDBYTE for 68K 327
  UNSIGNEDBYTE for PowerPC 219
  UNSIGNEDLONG for 68K 328
  UNSIGNEDLONG for PowerPC 220
  UNSIGNEDWORD for 68K 328
  UNSIGNEDWORD for PowerPC 219
Dead-strip Static Initialization Code option, 68K Linker panel 294
Dead-strip Static Initialization Code option, PPC Linker panel 198
deadstripping unused code 216, 320
debug strings 360
Debug_Version 385
Debugger 358
debugger 353–??, 359–365
  advanced topics 367–380
  breaking into 415
  code resources
    code resource debugging 363–365
  cross debugger 353–??, 353–??
  described 27
  exceptions 374
  new 369
  profiled code 379–380
  shared libraries 361–362
  shared library limitations 361
  SOM 375–377
  threads 378–379
  See also Debugger User Guide
debugger nub
  defined 352
Debugger Preferences 356–358
Debugger Target Settings panel 204–205, 298–299
Debugger() 358, 359, 360, 414
DebuggerINIT 353
debugging 31, 39, 361
  C++ 367–374
  C++ constructors 369
  C++ exceptions 374
  Drag Manager 412
  inline functions 372
  profiled code 379
  reentrant code 364
  shared library without application SYM file 362
  SOM 375
  static constructors 371
  static objects 371
  template classes 373
  templates 373
debugging PowerPlant 367–374
DebugNew 381–389
  and DebugStr() traps 384
  disabling 385
  memory overhead 387
  NEW macro 384

setting up 382
with MW ZoneRanger 387
DebugNewReportLeaks() 385
DebugNewSetErrorHandler() 384
using 387
DebugNewValidateAllBlocks() 384
DebugNewValidatePtr() 384
DebugStr() 358, 359, 360, 414
with DebugNew 384
delete (C++). See destructors
derez 159
destructors (C++) 372
development tools 25–28
device driver 102
directives, assembler
entry 240
fralloc 241
frfree 241
machine 242
smclass 243
Disassemble 160
Disassemble Exception Tables option, 68K Disassembler panel 290
Disassemble Exception Tables option, PPC Disassembler panel 195
Display Dialog option 119, 265
Display Dialog option, 68K Target panel 267
Display Dialog option, PPC Target panel 181
Display Manager aware 73, 178, 259
Display Notice event, handling 73, 178, 259
`divs.l` assembly statement 349
`divsl` assembly statement 349
`divu.l` assembly statement 349
`divul` assembly statement 349
documentation architecture 18
doesActivateOnFGSwitch 72, 177, 258
Don't Inline 373
Don't Step Into Runtime Support Code 369
DOUBLE 221, 329, 334, 335, 337
`double` size 210, 285, 308
drag and drop 49
Drag Manager
debugging 412
dynamic library 75
dynamically linked libraries, *see* shared libraries

E

editing code 30
See also IDE User Guide
`__eieio()` 244
emulation 413
`entry` assembly statement 240
entry directive 240
entrypoints 399
Enumerations 224, 332
on PowerPC 224
storage of enumerated constants on 68K 332
error
bus 415
exception handling 398
.exp file 92
Expand Uninitialized Data option, CFM68K panel 203, 297
Expand Uninitialized Data option, PPC PEF panel 203
export pragma 94
Export Symbols option, CFM68K panel 200, 296
Export Symbols option, PPC PEF panel 200
Export Symbols pop-up menu 93
exporting symbols 92
`extb.l` assembly statement 349
EXTENDED 329, 335, 337
Extended Resource option 113
Extended Resource option, 68K Target panel 267
`extended80` 316
extern keyword 75

F

`__fabs()` 245
Far Data option 311
Far Data option, 68K Processor panel 286
Far Data Threshold option, CFM68K panel 298
`far` keyword 311
Far Method Tables option, 68K Processor panel 287
Far String Constants option, 68K Processor panel 287
`far_data` pragma 311

Fast Link option, 68K Linker panel 293
fat
code resource 103
Fat applications 34
File Mappings 159
File Name field 96, 115
File Name option, 68K Target panel 257, 265, 270, 271, 272, 274, 275
File Name option, PPC Target panel 176
`float` size 210, 308
floating point formats for PowerPC 210
Floating Point option, 68K Processor panel 281
floating-point
formats 285
floating-point formats 210, 307
Floating-point numbers
and PowerPC calling conventions 226
Floating-point types 328
Floating-point unit 329
`__fnabs()` 245
Force Indirect Access option, CFM68K panel 297
fp68k.lib 282
FPSCR 246
Fragment Name option, CFM68K panel 203, 298
Fragment Name option, PPC PEF panel 203
`fralloc` assembly statement 236, 344
fralloc directive 241
`__fres()` 246
`frfree` assembly statement 236, 344
frfree directive 241
`__fsel()` 246
`__fsqrte()` 246
Full Path in SYM Files option, 68K Linker panel 197, 292
Full Path in SYM Files option, PPC Linker panel 197
function initialization 344
function level assembly 230, 340
Function results
Result passing conventions 336
Functions
and PowerPC calling conventions 227
Functions results 335

G

Generate A6 Stack Frames option, 68K Linker panel 292
Generate Link Map check box 197
Generate Link Map option, 68K Linker panel 292
Generate Link Map option, PPC Linker panel 197
Generate Profiler Information option, 68K Processor panel 283
Generate Profiler Information option, PPC Processor panel 190
Generate SYM File option, 68K Linker panel 196, 292
Generate SYM File option, PPC Linker panel 196
generating code
for specific 68K processors 313–318
GestaltEqu.h 395
GetDiskFragment() 361
getFrontClicks 72, 177, 258
GetMemFragment 361
GetMemFragment() 361
GetSharedLibrary 361
GetSharedLibrary() 361
global data
in code resources 123
Global Data Alignment option, CFM68K panel 297

H

header files
for code resources 124
Header Type 108
Header Type option, 68K Target panel 268
Header Type option, PPC Target panel 183
headers 396
headers, C/C++ 396

I

IDE described 25
IEEE floating-point standards 316
Import “weak” 393
import libraries, *see* shared libraries
Initialization option, CFM68K panel 199, 296
Initialization option, PPC Linker panel 199
Initialize Before command 394

INLINE 324
inline assembler
    asm blocks not supported 230, 340
    comments 234, 342
    defining a function 232, 341
    directives 240–243, 346–349
    for 68K 339–349
    for PowerPC 229–247
    function level support 230, 340
    instructions 229, 230, 339, 340, 349
    labels 234, 342
    local variables 235, 344
    operands 237
    preprocessor use 235, 343
    special PowerPC instructions 231
    stack frame 236
    syntax 230, 340
    using for 68K 339–345
    using for PowerPC 229–240
    See also intrinsic functions
INLINE directive
    example of procedure declared using 325
inline functions 372
    debugging 372
Inside Macintosh 325
Installing 404
installing CodeWarrior 24
int size 209, 307
INTEGER 218, 326, 334, 335, 337
    and arithmetic on 68K 326
    and arithmetic on PowerPC 218
integer formats 209, 306
integer formats for PowerPC 209
Integer types 218, 326
InterfaceLib 391, 393
interfaces 396
interfaces, Pascal 396
interrupts 364
    and debugging 413
intrinsic functions
    described 243
    for PowerPC 243–247
    See also inline assembler 243
IP address 355
is32BitCompatable 72, 177, 258
isDisplayManagerAware 73, 178, 259
isHighLevelEventAware 73, 177, 259
isStationeryAware 73, 178, 259
__isync() 244

J

Jasik's The Debugger 292
JSR (Jump to SubRoutine) 324

K

kill command 356

L

labels in inline assembler 234, 342
__labs() 245
language extensions for 68K 302–305, 324–325
launch application
    automatically 372
LBroadcaster 368
leaks.log 385, 386
__lhbrx() 245
libraries
    68K 63, 84
    A4 76
    A5 76
    adding 38
    adding to project 49
    CFM68K 66, 88
    changing options 83–85
    choosing 62
    creating 77–83
    creating 68K 77
    creating PowerPC 80
    creating shared 89–95
    for code resources 121
    for shared libraries 96
    MSL names 421
    naming 421
    naming conventions 62
    Pascal 66, 67
    PowerPC 65, 84, 89
    shared 75–85, 87–98
    SIOUX 63
Library Folder ID option, PPC PEF panel 203

Library SubFolderID option, CFM68K panel 203, 298
Link Column 55, 61
link errors 51
Link Mode option, PPC Linker panel 198
link order 216, 320
Link Single Segment option 111, 114
Link Single Segment option, 68K Linker panel 293
linker
.a files 216, 320
.o files 216, 320
and executable files 320
described 27
for 68K 320–321
for PowerPC 215–216
multiply defined symbols 216, 320
other documentation 208, 301, 323
Linker option, Target Settings panel 174, 254
linking 30
See also IDE User Guide
local variables in inline assembler 235, 344
localAndRemoteHLEvents 73, 178, 259
Locked 119, 182, 268
log window 360
`long double` size 210, 286, 309
long long
type 238
`long long` size 210, 307
`long` size 209, 307
LONGINT 218, 326, 334, 335, 337
and arithmetic on 68K 326
and arithmetic on PowerPC 218
LONGREAL 221, 329
`longreal` size 286
__lwbrx() 245

M

M68000PM/AD M68000 Family Programmer's Reference Manual 339, 340
Mac OS 413
Gestalt Manager 308, 309, 312, 314
Mac OS 68K Linker 254
Mac OS Merge 163–168
Mac OS Merge Linker 53
Mac OS PPC Linker 43, 81
Mac OS Support files 391–406
MacApp 293, 361
MacHeaders prefix 397
`machine` assembly statement 242, 349
machine directive 242
Macintosh
Mac OS routines 325
Macintosh system requirements 23
MacOS
and 68K registers 338
MacOS.lib 391, 396
MacsBug 292, 361, 415, 418, 420
MacsBug Symbols option, 68K Linker panel 292
MacTCP 355
Main option, CFM68K panel 199, 296
Main option, PPC Linker panel 199
Make 51, 62
Make Strings ReadOnly option, PPC Processor panel 190
makefiles 29
Math coprocessor 329
MC68020 processor 34, 262, 314
MC68881 FPU 315
Merge Compiler Glue into Segment 1 option, 68K Linker panel 293
Merge To File option 119
Merge to File option, 68K Target panel 267
Merge to File option, PPC Target panel 181
MetroNub 352
MetroNub Extension 354
MetroNub Settings panel 360, 361
Metrowerks Standard Libraries 75
Metrowerks Standard Libraries *See* MSL
Minimum Heap Size 60, 70
Minimum Heap Size option, 68K Target panel 261
Minimum Heap Size option, PPC Target panel 179
Mixed Mode Manager 413
modifying
target settings 169–171, 249–251
MPW C Calling Conventions option, 68K Processor panel 283
MPW Rez 158
MPW Shell 152

MPW Tools 131–152, 155
choosing libraries for 147–150
running 152
writing 150
MSL 63, 75
described 28
See also MSL C Reference, MSL C++ Reference
MSL CustomCRStartup.Lib 121
MSL Runtime68K.A4.Lib 121
MSL Runtime68K.Lib 64
MSL RuntimePPC.DLL 67, 123
`muls.l` assembly statement 349
multi-fragment applications 401
multiple processors 406
multiple symbols and linker 216, 320
multiple targets 40
creating Fat applications 40
multiprocessor development 406
multi-segment code resource 110
`mulu.l` assembly statement 349
MWCRuntime.lib 122
MWDebug preferences
Stop for DebugStrs 358
MWException.cp 374

N

naming conventions 421
network
configuring 355
requirements 354
None 95
nub. <$Italic<$Default Para Font
number formats
floating-point 210
for 68K 306–309
for PowerPC 208–210
integers 209

O

ObjectSupportLib 353
Old Definition option, CFM68K panel 202, 297
Old Definition option, PPC PEF panel 202
Old Implementation option, CFM68K panel 202, 297
Old Implementation option, PPC PEF panel 202
Only Show Operands and Mnemonics option, 68K Disassembler panel 290
Only Show Operands and Mnemonics option, PPC Disassembler panel 194
onlyBackground 72, 177, 258
Onyx Technology 419
Open Transport 354
operands in inline assembler 237
optimizing
for 68K 312–313
for PowerPC 213–214
inline assembly disables 231, 341
register coloring 213, 312
Ord4() function 219, 327
Output Directory option, Target Settings panel 174, 254

P

PACKED Arrays 222, 330
PACKED arrays of CHAR 223, 331
and C-style string constants 223, 331
and Pascal strings 223, 331
PalmOS library option, 68K Target panel 270
`parameter` pragma 304
Parameters and PowerPC calling conventions 225
Pascal
interfaces 396
Pascal interfaces 396
Pascal Language Reference 325
Pascal strings 222, 223, 331
PC-Relative Strings option, 68K Processor panel 287
`pcrelstrings` pragma 305
Peephole Optimization option, PPC Processor panel 192
PhotoShop plugins 363
Photoshop plugins 102
picking a target 172, 252
Port ID 355
Post-Linker option, Target Settings panel 174, 254
PowerMac Debug Services 353
PowerPC 122, 219
and 128-bit floating-point numbers 221

and CodeWarrior compiler 226, 227
Calling conventions
and composite parameters 226
and floating-point numbers 226
and functions 227
and parameters 225
and registers 226
code example 227
overview 225
code resources, calling 128
DOUBLE 221
enumerations 224
INTEGER 218
integer data types 218
LONGINT 218
LONGREAL 221
PACKED array of CHAR 223
Pascal/STRING 222
REAL 221
records 224
Sets 223
SINGLE 221
UNSIGNEDBYTE 219
UNSIGNEDLONG 220
PowerPC accelerated code resource 103
PowerPC intrinsics *See* intrinsic functions
PowerPC private code resource 103
PowerPC settings panels *See* settings panels
PowerPlant
debugging 367–374
viewing source code 372
PowerPlant Constructor 155
PPC (PowerPC) runtime library 397
PPC Disassembler panel 192–195
PPC Linker panel 195–199
PPC PEF panel 199–203
PPC Processor panel 187–192
PPC Project preference panel
code resource 114
shared library 95
PPC runtime library
installing initialization and termination entrypoints 404
PPC Target panel 175–183
PPCTraceEnabler 353
pragma
for 68K 319
for PowerPC 214
overload 320
precompiled headers 396, 397
preference panels
68K Project 114
68K project 95
PPC Project 95, 114
x86 Settings 355
Preferences
Debugger 356–358
preferences
Catch 68K DebugStrs 358, 360
Catch PowerPC DebugStrs 358, 360
Keep program in background while stepping 357
launch applications automatically 372
stepping into runtime code 369
Stop for DebugStrs 361
Preferred Heap Size 60, 70
Preferred Heap Size option, 68K Target panel 261
Preferred Heap Size option, PPC Target panel 179
Prefix File 397
prefix file 397
Pre-Linker option, Target Settings panel 174, 254
Preload 119, 182, 268
preprocessing 31
See also IDE User Guide
preprocessor, using in inline assembler 235, 343
printf 63
private code resource 103
ProfilerInit()
warning 380
ProfilerTerm()
warning 380
Program window
threads pop-up menu 378
Project Inspector 46, 51
project stationery 33, 35, 170, 250
Project Type option, 68K Target panel 255
Project Type option, PPC Target panel 176
projects
adding libraries 49
adding resource files 156

applications 35–74
building 39
choosing options 68–74
complex 40
configuring options 38
creating 35–74
creating libraries 77–83
making 51
making backup copy 42, 57
naming 38
renaming 51
resource files 158
resources 155–160
stationery 80
subprojects 40, 55
Protected option 119, 182, 268
Pull 335
Purgeable 119, 182, 268
Push 335
putchar 63

Q

Quark XTensions 363

R

REAL 221, 329, 334, 335, 337
RECORD 337
Records 224, 332
field alignment for 68K 332
on PowerPC 224
reference variables (C++) 369
register coloring optimization 213, 312
register variables for 68K 311
register variables for PowerPC 212
Registers
PowerPC and calling conventions 226
registers
floating-point 316
passing arugments in 304
variables 212, 311
Relocate Dwarf Info 195
Remote IP Address 355
Remove Selected Items 59
requirements *See* system requirements
Res Type option, 68K Target panel 266, 275
Res Type option, PPC Target panel 181
ResEdit 155, 266
ResID 108
ResID field 116
ResID option, 68K Target panel 267, 275
ResID option, PPC Target panel 181
Resorcerer 155
resource compiler
decompilation 159
resource files
adding to projects 156
creating 159
project folder 158
Resource Flags 117
Resource Flags option, 68K Target panel 267
Resource Flags option, PPC Target panel 182
Resource Name field 115
Resource Name option, 68K Target 266, 275
Resource Name option, PPC Target 180
resources
'CODE' 34
'rseg' 34, 262
'SIZE' 60
adding 155–160
ResType 108
ResType field 116
resume events, handling 72, 177, 258
Rez 159
Rez compiler 156
`__rlwimi()` 247
`__rlwinm()` 247
`__rlwnm()` 247
RSEG Application item, 68K Target panel 263
`rtd` assembly statement 349
runtime breakpoints 359, 360
Runtime Libraries 397
MSL CustomCRStartup.Lib 121
MSL Runtime68K.A4.Lib 121
MSL Runtime68K.Lib 64
MSL RuntimePPC.DLL 67, 123

S

safe fat code resource 103

SANE 282
SANE.h 316
Save A Copy As 57
Save A Copy As command 42
Save Project Entries, Target Settings panel 174, 254
scan 63
SegType field 116
SegType option, 68K Target panel 267
selecting a target 172, 252
SET 338
Set Current Target 51
`SetApplLimit()` 71, 180, 263
SetCurrentA4 363
SetCurrentA4() 363
`__setflm()` 246
Sets
  isolating one member of 332
  on 68K 332
  on PowerPC 223
settings panels
  68K Disassembler 288–290
  68K Linker 291–294
  68K Processor 278–288
  68K Target 255–278
  CFM68K 294–298
  Debugger Target 204–205, 298–299
  for 68K 251–294
  for PowerPC 171–??
  PPC Disassembler 192–195
  PPC Linker 195–199
  PPC PEF 199–203
  PPC Processor 187–192
  PPC Target 175–183
  selecting 170, 250
  Target Settings panel 172–175, 252–255
  See also target settings
Share Data option, CFM68K panel 203, 297
Share Data option, PPC PEF panel 203
Shared Libraries
  special considerations for binaries 392
  using applications 394
Shared libraries 361
shared libraries 75–85, 87–98
  CFM68K 88
  changing name 96
  changing options 95–96
  choosing libraries 96
  creating 89–95
  creating CFM68K 89
  creating Fat 91
  creating PowerPC 90
  Creator 96
  debugger limitations 361
  debugging 361–362
  exporting symbols 92, 92–95
  PowerPC 89
  shipping considerations 392
  support files 392
  Type 96
shared library
  debugging without SYM file 362
`short double` size 210, 308
`short` size 209, 307
Show Code Modules option, 68K Disassembler panel 289
Show Code Modules option, PPC Disassembler panel 193
Show Data Modules option, 68K Disassembler panel 290
Show Data Modules option, PPC Disassembler panel 194
Show Name Table option, 68K Disassembler panel 290
Show Source Code option, 68K Disassembler panel 290
Show Source Code option, PPC Disassembler panel 194
Show SYM Info option, 68K Disassembler panel 290
Show SYM Info option, PPC Disassembler panel 195
`signed char` size 209, 307
Simple type parameters 333, 334
SINGLE 221, 329
single-segment code resource 110
SIOUX 63
68881 Codegen option 308, 315
SIZE Flag 60
'SIZE' flags 71
  acceptAppDiedEvents 72, 177, 258

acceptSuspendResumeEvents 72, 177, 258
canBackground 72, 177, 258
doesActivateOnFGSwitch 72, 177, 258
getFrontClicks 72, 177, 258
is32BitCompatable 72, 177, 258
isDisplayManagerAware 73, 178, 259
isHighLevelEventAware 73, 177, 259
isStationeryAware 73, 178, 259
localAndRemoteHLEvents 73, 178, 259
onlyBackground 72, 177, 258
useTextEditServices 73, 178, 259
'SIZE' Flags option, 68K Target panel 258
176
`smclass` assembly statement 243
smclass directive 243
SOM 375
Stack 335
stack
viewing contents 420
stack frame in inline assembler 236
Stack Size 71, 180, 263
Stack Size option, 68K Target panel 263
Stack Size option, PPC Target panel 180
stand-alone assembler 27
See also Assembler Guide
Startup Code option, 68K Target panel 259
static constructors
debugging 371
preventing execution 371
static destructors 372
stationery 33, 35, 80
stationery aware 73, 178, 259
Step Into command (debugger)
constructors 370
stepping
into runtime code 369
__sthbrx() 245
stop command 356
stopping execution
at runtime 359
under program control 359
Store Static Data in TOC option, PPC Processor panel 190
STRING 337
string literals
PC-relative 305
Struct Alignment option, 68K Processor panel 280
Struct Alignment option, PPC Processor panel 188
Struct Alignment preferences panel option 222, 224, 330, 332
`struct` assembly construct 343
__stwbrx() 245
subprojects 40, 55
Suppress Warning Messages check box 198
Suppress Warning Messages option, 68K Linker panel 293
Suppress Warning Messages option, PPC Linker panel 198
Suppress Warning of Redeclared Types option 157
suspend events, handling 72, 177, 258
SYM file 361, 362, 363, 372, 373
Sym Name field 115
Sym Name option, 68K Target panel 265, 275
Sym Name option, PPC Target panel 180
symbols
exporting 92
multiple linker 216, 320
`__sync()` 244
SysBreak() 359, 360, 415
SysBreakStr() 358, 359, 360, 415
System Folder 355
debugger requirements 354
System Heap 119, 182, 268
System Object Model 375
system requirements 23
Macintosh 23

T

Target Name option, Target Settings panel 174, 254
target settings 249–294
dialog box 169, 249
modifying 169–171, 249–251
overview 169–171, 249–251
See also settings panels
Target Settings dialog box 169, 249
Target Settings panel 172–175, 252–255
target, selecting a 172, 252
targets

assigning files to 46
creating Fat applications 40
multiple 40
Targets category tab 42
TCP/IP 355
template classes
debugging 373
Termination option, CFM68K panel 199, 296
Termination option, PPC Linker panel 199
Text Edit services, using 73, 178, 259
The Debugger 292
The Debugger, Jasik's 418
32-bit clean applications 72, 177, 258
Threads pop-up menu 378
__throw() 374
ToolServer 152
Treat Multiple Definitions as Warnings option, PPC Linker panel 199
troubleshooting
bus error variables 415
debugger crashes 413
Drag Manager 412
exception handlers 411
Jasik's The Debugger 418
QC 419
stack crawls 420
try block, debugging 374
Type 108
Type field 69, 96, 116
Type option, 68K Target panel 266, 272
Type option, PPC Target panel 167, 179
types
long long 238
Types.h 316

U

Universal Headers 391, 395
Universal Interfaces 395
Universal Procedure Pointer
and debugging 413
`unsigned char` size 209, 307
`unsigned int` size 209, 307
`unsigned long long` size 210, 307
`unsigned long` size 209, 307
`unsigned short` size 209, 307
UNSIGNEDBYTE 219, 327
UNSIGNEDLONG 220, 328, 334, 337
equivalency to 4-element packed array of CHAR 220
UNSIGNEDWOR 334
UNSIGNEDWORD 219, 328, 337
UPP
and debugging 413
Use #pragma 94
Use .exp file 93
Use Extended Mnemonics option, PPC Disassembler panel 194
Use FMADD & FMSUB option, PPC Processor panel 191
USES 396
useTextEditServices 73, 178, 259

V

variable allocation for 68K 310
variables
register 212, 311
virtual function dispatch tables 287

W

WDEF 101
weak, import 393
what's in this manual 19
what's new 16
where to learn more 21
Window Definitions 101
Windows 95 354
Windows NT 354

X

`x80tox96()` 316
x86 Settings
Port ID 355
x86 Settings preference panel 355
`x96tox80()` 316
XCMD resources 363

CodeWarrior

Targeting Mac OS

Credits

writing lead: Alisa Dean

other writers: Carl B. Constantine and Stephen Chernicoff

engineering: Berardino Barratta, Dan Podwall, Fred Peterson, Benjamin Koznik, Marcel Achim, Glenn Meter, Lawrence You

frontline warriors: Richard Atwell, John Daub, Patrick Sullivan, Derek Saldana, David Blache, Kevin Bell, Matt Henderson, Ron Liechty, Metrowerks Support, and CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows